Forschungsinstitut für anwendungsorientierte Wissensverarbeitung
an der Universität Ulm
Institutsleiter: Prof. Dr. Dr. F. J. Radermacher

Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach

Dissertation zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Informatik der Universität Ulm

UNIVERSITÄT ULM · SCIENDO · DOCENDO · CURANDO ·

Christian Schlegel

aus Marbach / Neckar

2004

Amtierender Dekan:	Prof. Dr. F. W. von Henke
1. Gutachter:	Prof. Dr. Dr. F. J. Radermacher
2. Gutachter:	Prof. Dr. F. W. von Henke
Tag der Promotion :	20. April 2004

Abstract

Mobile robots in general and service systems in particular are often required to operate in the same environment as humans, sometimes even along with humans. Such systems have to be able to perceive their environment and to react to perceived changes appropriately. The goal often is not anymore maximum efficiency but reliable and flexible execution of various tasks even when the environment is dynamic.

As an increasing number of components is needed to perform different tasks and to cope with a set of situations, two issues are raised immediately.

The first issue is related to restricted resources on board a mobile platform. Thus, one needs approaches that are balanced in terms of resource consumption, reactivity and achieved results.

The second issue is related to managing the overall complexity. The capacity to perform various tasks depends on the integration of a substantial amount of skills. Although three-layer architectures evolved as standard for taskable robotic systems, the lack of standards for components and for the communication between components makes it hard to implement a complex robotic architecture.

Thus, the goal of this thesis is twofold. The first goal is to make the development of taskable robots more efficient, and the second goal is to increase their capabilities by providing balanced skills that exploit the advantages of loosely coupled components.

The first goal is met by a software concept tailored to the needs of taskable robots. The complexity issue is tackled by means of *communication patterns* that provide a fixed semantics for component interactions. The *dynamic wiring* pattern is the basis for making the control flow and the data flow configurable at runtime from outside a component as needed to fully exploit the power of three-layer architectures.

The second goal is deepened with respect to navigation with a focus on mobility since mobility is a fundamental and pivotal capability for many service robotic applications.

The verification of the approach is achieved by an implementation of a full-fledged system that is able to perform various complex fetch and carry tasks as well as recognition tasks in a dynamic test environment.

Acknowledgments

A thesis can never be completed without a supportive environment. It is therefore my pleasure to thank everyone who directly or indirectly contributed to this work in some way or other.

First of all, I thank my thesis advisor Prof. Franz-Josef Radermacher. While he encouraged me numerous times to come to an end with my work, he was still patient enough to wait until I finally did. I owe Prof. Radermacher many thanks for his continuous encouragement and his persistent interest in my work. He always gave me the freedom to pursue my own ideas and never lacked the faith in my research.

I am also grateful to Prof. Friedrich von Henke for his spontaneous agreement to act as referee. In particular, I appreciate the preparedness to instantly read through earlier drafts of this thesis and his invaluable support to meet the tight schedule to finally complete the thesis in due time.

Furthermore, I thank Prof. Günther Palm and Prof. Jörg Kaiser for their instant readiness to join my thesis committee.

I owe many thanks to Thomas Kämpke. He crucially influenced my academic development and always pushed me to thorougly revise all my ideas. He was the endless source of inspirations that helped me to avoid many side-tracks and dead-end roads in my research work. Thomas Kämpke always insisted on doing research besides the bread-and-butter job of meeting project deadlines. Most important and being of extreme value, he created an atmosphere and a setting that provided room for doing so. I owe Thomas Kämpke also many thanks for the mental support during the final stage of this thesis.

I had the privilege to continuously work in the exciting field of autonomous systems. I greatly appreciate that chance provided by Prof. Radermacher, Peter Spiertz and Thomas Kämpke.

Finally, I would like to express my thanks to my colleagues at FAW and at the University of Ulm for many fruitful discussions. In particular, I would like to thank all my colleagues in the Autonomous Systems group at FAW, notably Boris Kluge and my former colleague Robert Wörz.

Last but not least, I am deeply grateful to my family. There are no words to express what I owe my wife Alexandra for sharing all the ups and downs. Especially during the final stages, she relieved me of all everyday duties and even cancelled many of her activities. Although she as well had to bear the brunt of an unsettled future, she always encouraged and motivated me even during the catchiest periods. Timo, you taught me what is really important about life.

Contents

Chapter 1

Introduction

The field of robotics has gained more and more attention over the last years. In particular, the development of *service robotics* is identified as a tremendous challenge. Many potential applications require a robot to operate in the same environment as humans, sometimes even along with humans. Such systems have to be able to perceive their environment and to react to perceived changes appropriately. The goal often is not anymore maximum efficiency but reliable and flexible execution of various tasks even in a dynamic environment.

In reference to the best available knowledge that requires the integrated use of symbolic and subsymbolic mechanisms of information processing with both forms of information processing interacting tightly. Subsymbolic mechanisms ensure flexibility and reactivity even in dynamic environments since they typically allow for short cycle times. Symbolic approaches afford goal-oriented activities but typically require time-consuming calculations.

The state-of-the-art architecture for such systems is a three-layer architecture. In this work, the lowest so-called *skill layer* consists of components that operate on the level of sensors and actors. These mainly form control loops and, for example, build a map of the environment or ensure collision free motion. The medium so-called *sequencing layer* is responsible for the situation dependent selection and configuration of components and also coordinates the task execution by synchronizing the execution progress with a discrete description of desired plots. Finally, the uppermost *deliberation layer* comprises time-consuming algorithms like the symbolic task planning.

The organizational strength of a three-layer architecture is founded in its fundamental support of a situation dependent deployment of the robot's capabilities. That is of particular importance since service robotic applications have to cope with many different challenges while executing a task. These are typically such that, for example, a single instance of a motion control algorithm is not able to handle all kinds of motion control tasks efficiently. A three-layer architecture develops its full power only in case it has a substantial set of skills at its disposal. Furthermore, these skills have to be prepared to get recombined to form different behaviors. Without being able to configure and recombine the skills such that they are adapted to the encountered situation or task at issue, one is typically not able to perform even a standard *fetch and carry* task.

As an increasing number of components is needed to perform different tasks and to cope with a set of situations, specific demands on the various components of such systems are imposed and two issues are raised immediately. The first issue concerns restricted resources on board the robot. Thus, it is not possible to consider each component in isolation. In most cases, simply combining approaches, that are already available for subtasks, results in far too resource intense solutions. In fact, one has to devise new methods for subtasks such that their interplay results in an overall solution that is balanced in terms of resource consumption, reactivity and achieved results, for example.

The second issue is related to managing the complexity of service robot systems. Successfully implementing a three-layer architecture is tightly related to a software architecture. Without a software concept, one hardly achieves an interface semantics that ensures the level of decoupling that is required for recombining skills to different behaviors. Task execution in a three-layer architecture severely depends on the configurability of numerous interdependencies between distributed and concurrent activities. The decoupling concept of the software architecture is the crucial factor towards achieving the necessary modularity and flexibility of the implementation. Without adhering to the decoupling concept of the software architecture, the overall complexity of the system is hardly controllable and often even restricts the applicability of skills.

1.1 Thesis Outline and Contributions

This thesis is concerned with aspects of building *taskable* mobile robots. A *taskable* robot is not tailored to a specific purpose but is able to perform different tasks without requiring to rework the control system of the robot.

Chapter 2 gives a brief overview on the history of robot architectures and introduces the three-layer architecture as state-of-the-art architecture for taskable robots. A specific instantiation of a three-layer architecture is described that has been implemented on a mobile platform based on the contributions of this thesis.

Chapter 3 addresses mobility for robots in dynamic environments. The contribution is a balanced approach in terms of necessary computing power, achieved reactivity and deviation from optimality and completeness. It provides basic mobility in dynamic environments.

Chapter 4 provides an outline of a computation scheme for simultaneous localization and mapping. The computation scheme allows to acquire a preliminary map of an environment that can simultaneously already be used for localization and for navigation purposes. The type of acquired map allows to postpone time-consuming global map optimizations without loss of information.

Chapter 5 which is pivotal to this work, presents a software architecture that supports the development and implementation of taskable robots. It addresses the complexity issue by means of *communication patterns* that provide a fixed semantics for component interactions. The *dynamic wiring* pattern is the basis for making the control flow and the data flow configurable at runtime from outside a component as is needed with nearly any robotic architecture.

Chapter 6 shows the coaction of all contributions of this thesis. The workability of the approaches is illustrated by means of two demanding tasks, a pattern building task and a mail distribution task. Both tasks require a fully integrated system with balanced skills and can be executed only by tight interactions of all components.

Chapter 7 concludes with a brief summary and an outlook.

Chapter 2

The Architecture

2.1 Introduction

In general, an *architecture* is a description of how a system is constructed from basic components. An architecture also describes how components fit together to form the whole and it imposes constraints on how a robot system is to be structured. In the context of this chapter, the term *architecture* refers to the arrangement of the control mechanisms of a robot.

Of course, a great impact on the architecture of a robot stems from the purpose it is designed for. For example, wandering around without having to remember any external states results in an obvious control architecture. Thus, it is often assumed that the architecture is an immediate result that needs no further consideration. That view is fleshed out by the fact that any system can be described by some kind of architecture.

For a long time, it was assumed that there are no general principles that are worth to be abstracted from implemented systems. Nevertheless, it turned out, that implementations of robotic systems not only follow certain rules but that encountered limitations often are correlated to a certain type of architecture. Foremost, it has been recognized that there is a fundamental difficulty in bridging the gap between numerical sensor values and symbolic descriptions and between symbolic task specifications and their execution in real world. However, as soon as one wants a robot to execute various tasks in a changing environment, there is no obvious way to do without compact and abstract symbolic descriptions or to do without sensor based feedback. Although an architecture cannot close the fundamental gap between symbolic and subsymbolic mechanisms, it can arrange the interaction such that one can still take advantage from both worlds at least in certain settings.

2.2 A Brief History of Robot Architectures

A very well overview on the development of robot architectures is given in [98] which forms the basis for this survey. In earlier times, the *sense-plan-act* approach [116] was the dominant view. This classical approach comprises a *vertical* division of the control problem into several functional units as shown in figure 2.1. Each functional unit is responsible for a specific task. Information flows from sensors through all functional units to actuators and never in the reverse direction. The job of the perception system is to translate sensor data into a world model which is then used by the planning system. Executing a plan was considered simple compared to generating a plan. However, it turned out very soon that planning and world modeling is even harder than already assumed and that executing a plan without taking into account changes in the environment is an inadequate approach.

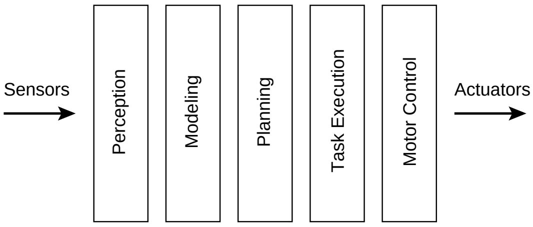


Figure 2.1: *Vertical decomposition.*

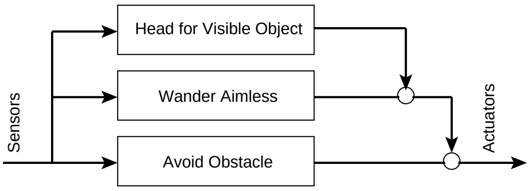


Figure 2.2: *Horizontal decomposition.*

The *subsumption architecture* [16] was a clear break with the *sense-plan-act* approach. As shown in figure 2.2, the control problem is tackled by a *horizontal* decomposition. Each layer implements one behavior. A behavior gets sensor input and produces actuator settings and is able to control the robot independently of the other layers. The advantage is that there is neither a centralized perception component that has to perceive everything nor is there a centralized planning that has to plan for every single step and contingency. In contrast, each behavior performs solely that kind of sensor processing and action generation that is needed with the behavior. Different behaviors are coordinated via suppression or inhibition that is one behavior can overwrite either the input or the output of another behavior. The design methodology is to stack behaviors of increasing complexity such that the output of a complex behavior gets overwritten as soon as a lower level behavior like obstacle avoidance gets active. The most prominent example of the subsumption architecture is the robot *Herbert* [24] that was programmed to find soda cans in an office environment.

However, there is an important issue with respect to complexity management which is not addressed at all by the subsumption architecture making it unsuitable as engineering methodology [67]. Behaviors cannot be designed independently since upper layers interfere with the lower layer behaviors. Even small changes to a behavior often require the redesign of the overall controller. The problem is further tightened by the fact that behaviors often cannot naturally be arranged in a hierarchy to apply the mechanism of suppression and inhibition. Decoupling behaviors by not maintaining shared states and using *the world as its best model* [18] causes severe limitations since similar states of the world can mean completely different things depending on the context in which they are encountered.

Consequently, new kinds of architectures have been proposed that address the problem of implementing goal directed behaviors by means of robust and reactive components. One class of approaches specifies both the goals and the methods required for their achievement by means of a specialized language that allows for more complex arbitration schemes. The specification is compiled into a decision network [85]. At execution time, the decision network maps sensor input to actions. Even though a decision network can represent conditional plans to handle contingencies, that results in a combinatoric explosion in the number of states. Furthermore, most of these systems have no notion of *sensing goals* and assume that all relevant states are perceived on the fly. Related work all addressing shortcomings of the purely reactive approach or its simplified arbitration scheme can be found in [125], [3] and [107].

A general drawback of these systems is that they are not *taskable* that is they are designed for a single task only and performing another task requires to rework the control system. At least three different groups came up with similar solutions of combining deliberation and reactivity [25] [53] [8]. All three approaches share the idea of using *three* main components: a reactive feedback control mechanism working with a short cycle time, a deliberative planning capability and a sequencing capability that connects the first two components. The importance of the sequencing capability was first

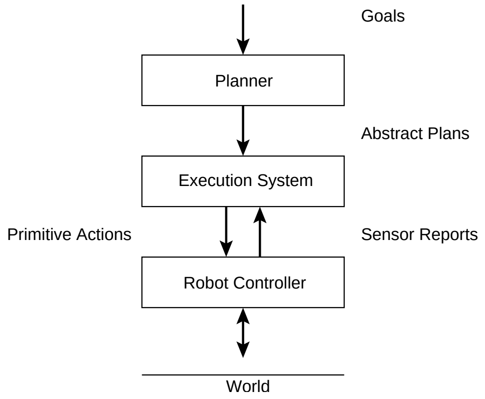


Figure 2.3: *The three-layer architecture [41]*

pinpointed in [41]. The three-layer architecture as presented in [41] is shown in figure 2.3. A good characterization on three-layer architectures can be found in [10].

The success of three-layer architectures results from its compliance with the empirical observation that control mechanisms for mobile robots can be divided into three distinct categories. The first category are reactive control algorithms called skills. These form the interface to the real world and ensure reactivity and responsiveness. However, these need to be configured depending on the task and the overall context and are composed to different more complex behaviors. Most important, they have to *fail cognizantly* [117]. Instead of being designed to never fail, they only have to be designed such that they can report each failure. This is a much more realistic assumption for algorithms operating a robot in real world and it forms the basis to take advantage from corrective actions of the other layers.

The second category are mechanisms for activating and deactivating skills, setting their parameters and composing them to behaviors. A sequencing component conditionally responds to the outcome of the executed behaviors. It expands the general plot of a task at execution time taking into account the encountered situation. This circumvents the combinatoric explosion that comes along with plots that try to consider any possible contingency in advance. In contrast to *plans*, *conditional sequencing* provides much richer control constructs to express procedural knowledge on how to execute a specific task. For example, *conditional sequencing* allows for *repair procedures* and can handle concurrent and interacting tasks. The sequencing component takes into account past and current states but no future states. For example, it does not check the effect of selecting a particular expansion by means of temporal projection.

The third category are mechanisms that reason in depth about future states and that generate plans, for instance. In principle, there are three different ways of interacting with the *deliberative layer*. The first option is to invoke the planner from the sequencing layer and to consider the planner as expert for specific tasks. The second option is to consider the planner as the only instance that provides to be executed tasks to the sequencing component. The third option is to have a two-way interaction between the sequencing layer and the deliberative layer.

The distinguishing feature of a three-layer architecture is that one can control real robots performing complex tasks even with a trivial deliberative layer and even with skills that cannot handle all situations. The sequencing component provides the necessary glue logic and is the place to store procedural knowledge that neither fits at the deliberative layer nor at the skill layer.

2.3 Instantiations of Three-Layer Architectures

In principle, there exist many different instantiations of this architecture. As soon as a mobile robot reaches a certain level of flexibility with respect to the tasks it can perform, one can find a structure according to a three-layer architecture, most often implicit and rarely explicit.

The sequencing layer of *SSS* [25] is based on subsumption. *Atlantis* [53] used the *RAP* system [41] before *ESL* [54] has been developed. The sequencing layer controls the task planner and it is tailored to programming convenience rather than to interacting with a symbolic planner. *3T* [9] first used the *REX/GAPPS* system [84] and switched to the *RAP* system. The *RAP* system was extended to handle continuous processes rather than atomic operators. The sequencing layer is controlled by the task planner.

Specialized languages [52] [17] to implement the skill layer did not show additional value compared to all-purpose programming languages. In contrast thereto, due to the needed control constructs, conditional sequencing is done much more efficiently with a specialized language. In principle, one can either extend a standard language or one can define a language in its own. The first approach is taken by *TCA* [139], *TDL* [141] and *ESL* [54], for example. Typically, the task refinement is *programmed* rather than *described* and the task refinement is just done as programmed without looking up behaviors matching a task specification. Thus, newly added alternatives are only considered when added to the respective control programs. The integration of a classical AI planner with a real robot based on *TCA* is described in [63].

The second approach is taken by the *RAP* system [41], *PRS* [56] and *RPL* [109], for example. The *procedural reasoning system (PRS)* is a general framework and it is argued in [73] that it can be adapted to robot control. *RPL* is tailored to handle execution flaws by means of plan repair rules. Thus, *RPL* provides means to support introspection to the structure of a controller as it is exploited by the *structured reactive controllers* [6]. The *RAP* system provides both, a declarative description of the preconditions and effects of a task and a procedural expansion of it in form of a task net. Such a unit is called a *reactive action package (RAP)*. A library holds *RAPs* and task net steps are matched against it at execution time. A newly added *RAP* is immediately available to all task nets. The *RAP* system has been used to successfully implement several robots [37] [55] [44] and has been extended several times [43]. A comparison of some robot programming languages can be found in [121]. An evaluation of architectures can be found in [118].

COLBERT is also called a language for conditional sequencing by its developers [94]. It is tightly integrated into the *Saphira* control architecture [96] and is based on finite state automatons with a fixed cycle time. However, these do not scale to more complex sequencing tasks. At the other end are *GOLOG* and its extensions [102] [57]. *GOLOG* is based on the situation calculus and provides very compact specifications of complex controllers. However, executing the output of a *GOLOG* program requires substantial glue logic [62]. For example, *Minerva* used a *GOLOG* component on top of the *structured reactive controllers* [147].

2.4 Design Decisions Made with Implementing a Three-Layer Architecture

The architecture of the system that is based on the contributions of this thesis is shown in figure 2.4. The major component with respect to control is the *agenda and interpreter* component. Many tasks are easily specified at an abstract level but their execution ultimately requires a complex interaction and sequencing of skills. It is based on a reimplementation of the *xRAP* system that is not publicly available [42]. The major difference to the *RAP* system are the enhanced control constructs as described in [42]. Furthermore, the restriction that only *primitive actions* can interact with skills is omitted. Thus, complex state transitions needed without deactivating skills can be shifted from the skill layer to the sequencing layer where powerful representations are available.

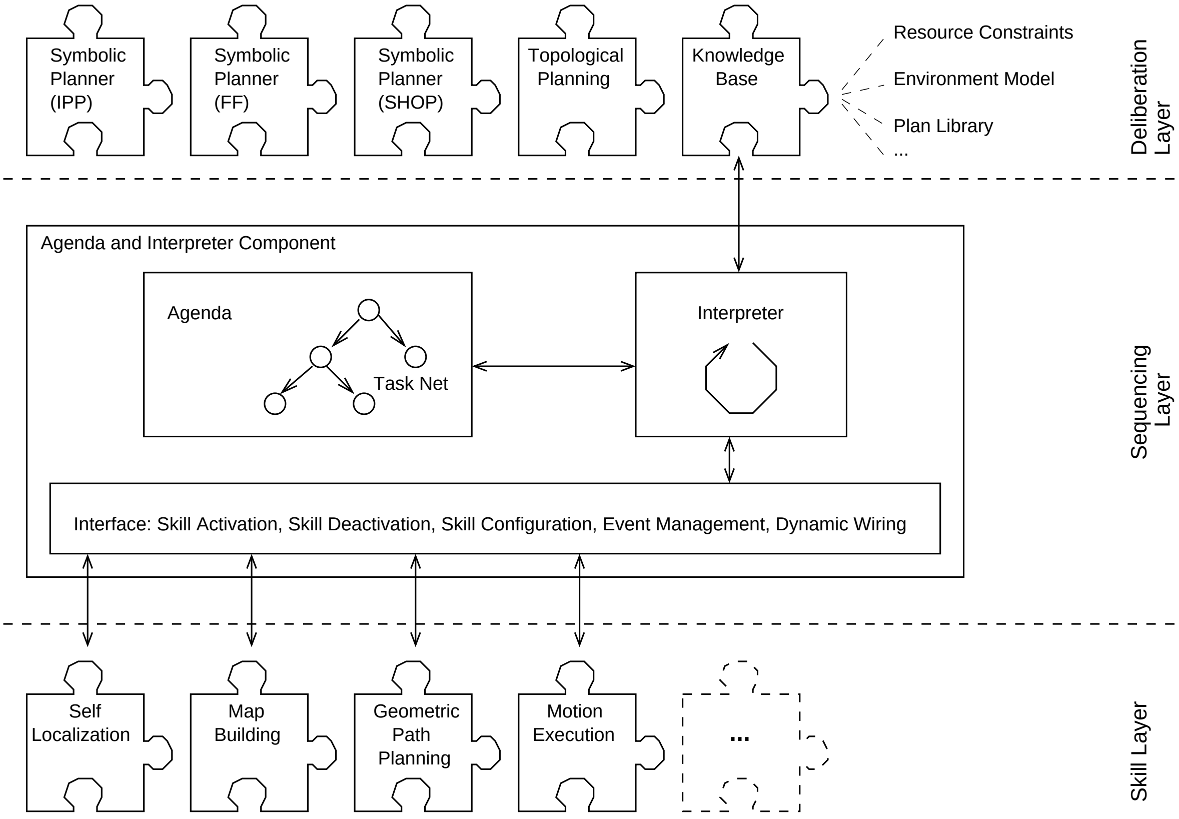


Figure 2.4: *The three-layer system architecture.*

All activities are represented on the agenda and new tasks have to be placed on the agenda for execution. Expansions of task nets can contain nodes whose execution results in invoking an *expert*. For example, a task planner at the deliberative layer is invoked in this way. A task planner is invoked for such tasks for which the task net formulation is not feasible due to the combinatoric explosion of states. For example, creating a certain pattern of colored discs on a table is done by moving to the table, invoking a sensing operation and then requesting the subsequent task net from the task planner.

In principle, the task planner is neither responsible for all activities nor has it to be able to handle any situation. Furthermore, plans are no rigid control regimes. Since we know for what kind of tasks the symbolic planner is invoked, we also know which contents of the knowledge base are relevant for the planning task. Only these are extracted and converted into a representation processable by the task planner. The result of the task planner is transformed into a task net. Task planners used comprise the *IPP* [93], the *FF* planner [70] and the *SHOP* family [114].

The task planner does not plan for any contingencies even though the representational power of the operators is severely improved compared to the first task planners [48]. Rather, execution flaws are handled by the sequencing component by means of task net transformation rules that insert appropriate activities at execution time. For example, the side effect of *moving* is that all pose labels of known objects are tagged as *inaccurate*. However, a grasping operation can be performed only on objects for which one knows the accurate pose. Thus, a transformation rule is triggered by the execution flaw and replaces the grasping operation by a sensing operation followed by the original grasping operation. The task nets, the transformation rules and the planner invocations contain the procedural knowledge on how to execute a task. Since these are represented in a *declarative* way, they can easily be extended and adjusted.

The knowledge base does neither hold a complete model of the environment nor is it always up-to-date. In fact, it provides markers as references to distributed representations. This allows to maintain skill specific aspects within the skills while the knowledge base provides the link between the distributed models. Further details can be found in [130] and [128].

The overall architecture depends on a rich set of skills and the configurability of the skills to behaviors. Furthermore, one needs a suitable approach to cope with the overall complexity of such a system. Only a standard for system decomposition allows to cope with the complexity issue. As stated in [31], in particular the lack of standards for components and the communication between components makes it hard to implement a complex robotic architecture. Thus, the software concept to implement such an architecture is attached great importance. Furthermore, behaviors have to be balanced in terms of needed computing power, achieved reactivity and deviation from optimality and completeness. However, they are allowed to *fail cognizantly*. Thus, balanced approaches for crucial skills of mobile robots are also attached great importance.

Chapter 3

Motion Control

3.1 Introduction

Motion control and obstacle avoidance are two of the most basic skills of a mobile platform. Their performance has a crucial impact on the achievable level of mobility. Of course, motion control and obstacle avoidance belong to the core disciplines of robotics. Although there exists a considerable amount of work on this topic, there is still a gap between reactive approaches and planning approaches and many approaches are still restricted to a certain setting. Reactive approaches ensure high mobility but suffer from many well-known shortcomings like local minima and are based on oversimplified assumptions like a circular robot shape. Planning approaches do not depend on those simplifications but suffer from impractical computational requirements in case they take into account any-shaped contours or kinematic and dynamic constraints.

Mobile service robots have to be able to reliably carry out tasks even in partially unknown and cluttered environments. Safe operation is becoming more and more important as many of the applications are in busy environments that cannot be highly engineered for the deployment of service robots. Thus, neither kinematic nor dynamic constraints can be ignored without putting a risk onto the environment and the mobile platform. Kinematic constraints restrict the possible movements of a platform and are introduced by the drive system and its configuration. For example, a tricycle cannot make lateral movements. Ignoring the kinematic constraints selects paths that finally cannot be executed. The platform inevitably deviates from the designated path which can result in a crash. Dynamic constraints can be ignored safely as long as the platform moves only at a very low speed since both stopping and adapting to any new heading can be done without requiring much free space. However, in case of higher speeds, for example, the required distance to come to a stop has to be considered.

Planning approaches are able to generate paths that take into account any relevant constraints. Of course, there are also approaches that guarantee to return a feasible trajectory in case there exists one. However, these calculations take so much time that the environment often has changed already till the then outdated result gets available. Thus, first generating a complete plan that takes into account any constraints and executing that after it got available is not an option. The ideal solution would be to integrate planning into the control loop but planning is far too slow thereto. In [21], it is shown that motion planning for a point in the plane with bounded velocity is NP-hard even when the moving obstacles are convex polygons that move with constant linear velocities without rotation. Foundations on motion planning in dynamic environments can be found in [51].

Thus, in a dynamic environment where one wants to drive with considerable velocities, one needs a different approach. One option is to reduce the requirements on the planning approach such that it can produce results in a reasonable time and thus, it can be integrated into a control loop. For example, instead of generating trajectories, one solely calculates feasible paths that do not specify any velocities. Further simplifications ignore the shape of the robot and even its kinematic constraints which considerably reduces the response time of the planning component. Of course, the more simplifications are made at the planning level, the less can one guarantee that the provided path can be accomplished by the mobile platform. Thus, one has to accept that the reactive execution deviates from the path specification of the planning component and that there are even settings in which such an approach is not able to reach a goal although a feasible trajectory exists. In principle, the task of the planning component is often reduced such that it only gives hints on suitable intermediate waypoints.

The less is covered by the planning component, the more responsibilities lie on the reactive execution component. It has to ensure safe operation under any circumstances. Although the reactive component already resigns all aspects that can be handled only with a global approach, the decision for the motion command of the next time step is still demanding. That is mainly due to the time constraints since the response time of the reactive component directly influences the cycle time of the control loop and thereby the allowed velocities and the tackled changes in the environment. In particular, the reactive component has to be able to in-line process the updated information about the current state of the environment. Thus, even reactive components introduce simplifications that also range from ignoring the kinematic and dynamic constraints to ignoring the robot shape.

All feasible approaches react to dynamic obstacles by sufficiently often sensing the environment and reacting to the perceived changes. Dynamic obstacles are only treated by means of a snapshot without considering their actual movement. Feasible approaches that take into account obstacle dynamics all require simplifications that are not compliant with the focus of this work. In principle, motion coordination is beyond the scope of this thesis.

3.2 The Problem

A severe limitation with many computationally feasible approaches is the assumption of a circular shaped robot. Although many platforms are circular in their basic configuration, that often does not hold anymore as soon as the platform has to carry a payload. That extends its contour so that a circular approximation of the contour is typically such large that the robot, for example, does even not fit anymore through regular sized doorways. Thus, approximating a non-circular robot with a circle is not a solution since that not only results in an extremely conservative avoidance behavior but also restricts the freedom of movement far too much. Often, the geometry of the robot is application driven and cannot be chosen arbitrarily.

3.2.1 Contributions

This chapter presents a fast local obstacle avoidance approach that takes into account kinematic and dynamic constraints. In contrast to other approaches, it can handle polygonal robot shapes and thus overcomes the assumption of a circular robot shape. It is called the *curvature distance lookup (CDL)* approach and was presented first in [126]. The shape can be adjusted at any time, for example, depending on the presence or absence of a payload. It can take into account obstacle information from almost any type of distance sensor and from various types of maps. Most important, even raw laser range scans can be used without requiring any preprocessing or data reduction step.

Furthermore, the interaction of the reactive *curvature distance lookup* approach with a simplified global path planner is presented. It allows to trade off reactivity and optimality without ever violating safety. Although this is achieved at the price of not being able to reach the goal in all cases, the approach is balanced in terms of necessary computing power, achieved reactivity and deviation from optimality and completeness. It is able to handle most of the standard settings in a reasonable way and always ensures a safe motion state even with a polygonal robot shape.

Using the proposed approach for standard motion control and obstacle avoidance tasks and having a full fledged path planner in store for those cases in which the proposed approach gets stuck, allows to cope with almost any setting in motion control and obstacle avoidance. In the line of three-layer-architectures, the presented approach covers the standard requirements on a motion control and obstacle avoidance component of mobile platforms that operate in standard indoor environments. One has to switch to other approaches only in case this approach gets stuck or for specialized and highly demanding maneuvers that can neither be generated nor executed by the presented approach. In those specialized and typically rare cases, one can accept both a high computational load and further constraints like very low velocities or a static environment at least for the time of the specialized maneuver.

3.3 Related Work

It is impossible to give an extensive survey on the rich body of work in the field of motion planning and obstacle avoidance. The only chance is to restrict the presentation to those contributions that address the very same subdomain. Thus, many approaches are not considered in detail since they are either not able to cope with a changing environment or require far too much processing power or focus on aspects that shall not be covered by the proposed approach. In particular, no approaches are considered that go beyond the assumption of a two-dimensional world as is for example needed for outdoor applications.

Although motion coordination is beyond the scope of this thesis, an interesting approach is the idea of *velocity obstacles* [40] that takes into account the obstacle dynamics. However, it is left open how to detect these in a typical setting of a service robot. Furthermore, the approach does not consider kinematic constraints and also assumes circular shaped robots. The output of the algorithm is a heading and a velocity and it selects only such a heading for which no collision can occur at any point of time from now on. Thus, the original approach gets into trouble as soon as it is surrounded by walls as is the case in any indoor environment.

Getting a motion estimate by a laser range finder in a typical service robotic scenario is tackled in [91]. It focuses on motion coordination and extends the velocity obstacle approach to incorporate reasoning about other agent's navigational decision processes. A practical solution is achieved but, of course, also only by making simplifying assumptions.

Early work in the field of local obstacle avoidance algorithms just determined a new heading without taking into account any kinematic or dynamic constraints. A prominent example is the *potential field* approach [90]. In its original idea, it is very sensitive to local minima [97]. Of course, the classical approach has been extended to either improve the driving behavior [88] or to overcome local minima [38]. However, both approaches assume a circular shaped robot.

Even though configuration space approaches [103] are able to deal with kinematic constraints, cluttered environments and any-shaped contours, they cannot be applied to dynamic environments due to the complexity of the mapping of obstacles into the configuration space. These approaches are mainly applied to manipulators with many degrees of freedom and a static environment. Contributions address the problem of the high-dimensional search space [87].

The *elastic band* approach [124] is a quite general framework for the execution of planned paths.

It represents a path by a curve in the configuration space with the properties of an elastic band. A once planned path is adjusted to sensed changes in the environment and its behavior can be illustrated by means of an elastic rubber band that gets deformed by moving obstacles while keeping some smoothness properties. It provides an interesting balance between the planning component and the execution component since the planned path gets adjusted incrementally. However, a path always maintains its topological properties so that the *elastic band* approach often results in a suboptimal path when there are moving obstacles or the path even gets invalid so that one needs replanning. However, the more fundamental problem of the *elastic band* approach is related to its representation of free space. Since the elastic band integrates the path modification into the control loop, the adjustment of the elastic band has to be done very efficiently. That, however, presumes knowledge about the free space around a path and again, a circular approximation is used to reduce the processing time. An extension of the *elastic band* approach to non-holonomic robots is presented in [89].

The *steering angle field* method [39] is one of the rare approaches that has been applied to a rectangular robot with a tricycle or equivalent kinematic. All steering angles are rejected that belong to paths colliding with an obstacle within a velocity-dependent time-interval. Different selection functions can be used to determine the final steering angle among a set of admissible ones. However, the approach solely computes a steering angle, and velocity control is an iterative process between the steering angle field module and a pilot module.

The *curvature velocity* method [140] and the *dynamic window* approach [47] are related to the steering angle field approach. In addition to kinematic constraints, these also take into account dynamic constraints. The *curvature velocity* method formulates the local obstacle avoidance problem as one of constrained optimization in velocity space. The *lane curvature* method [142] further improves that approach by overcoming problems due to the assumption of solely moving along circular arcs. The *dynamic window* approach also operates in velocity space. Dynamic constraints are used to reduce the velocity search space to values reachable within the next cycle. To allow for suitable cycle times, all these approaches assume circular shaped robots.

The *vector field histogram (VFH)* [12] transforms the two-dimensional environment into a one-dimensional histogram from which a heading for the motion of the robot is selected. It completely ignores dynamic and kinematic constraints. The approach was extended to the *VFH+* method [150] that provides a very rudimentary form of taking into account kinematic constraints. A heading of the *VFH* approach is rejected under some circumstances that depend on a to be defined minimum curvature. The approach still does not correctly consider kinematic constraints and also assumes a circular shaped robot.

There are many approaches that extend purely reactive approaches by integrating some kind of planning. The *VFH** [151] extends the *VFH/VFH+* idea by integrating a look-ahead step. Of course, the type of local minima that can be circumvented is directly related to the size of the look-ahead. Nevertheless, addressing the problem of local minima does not make any improvements with respect to handling robot shapes and kinematic constraints.

The *global dynamic window* approach [14] integrates the recomputation of a NF1 navigation function [99] into the objective function of the dynamic window approach. The navigation function is computed prior to the movement. Inside the control loop, only a narrow region of the free space connecting the goal and the current configuration is updated. The considered region is incrementally widened as soon as no path can be found due to obstacles. The objective function of the dynamic window prefers such motion commands that follow the gradient of the NF1 function at the current position of the robot.

The tight integration of the navigation function into the control loop has the advantage that the gradient of the NF1 function at the current position always directs along a shortest path to the goal.

However, the challenge is to update the gradient fast enough in case of perceived obstacles. Thus, only a narrow region is considered for updating within the control loop. Since the gradient has a substantial impact on the result of the objective function, an outdated gradient strongly directs the robot towards the not yet considered obstacle.

Another drawback of the tight integration of the navigation function into the objective function concerns the velocity control. The local gradient does not give any hints on the further characteristic of the path. Thus, even if the navigation function provides an optimal path, its execution often does not meet expectations since the robot is not able to follow the gradient once it accelerated due to free space. A typical situation is a hallway in which the robot accelerates and then misses the door to an office.

An advantage of the tight integration of a navigation function into the control loop is the option to perform the path planning in the configuration space as mentioned in [13]. Then, the local gradient at least takes into account the shape of the robot. However, one can avoid collisions only in case one always sticks to the planned path which makes the reactive component obsolete. If one relies on the reactive component, then that has to be able to take into account the robot shape. As soon as one makes the usual circular assumption there, one looses the advantages of a configuration space planner. By the way, even a partial update of a three-dimensional configuration space normally requires far too much time to be done within the control loop. Furthermore, one also would need an efficient way to translate perceived sensor data into configuration space obstacles.

Thus, the tight coupling in the above form is not feasible in case one wants to exploit it by means of a configuration space planner. The tight coupling is obsolete as soon as the reactive component is not able to consider the same constraints as the planning component. Planning in the configuration space and executing the path with an approximated circular shape, for example, inevitably results in deviations from the planned path. Then, however, one should not encounter better results with the above described tight integration than with the here proposed loose coupling that requires the reactive component to already handle the shape constraints and that exploits the idea of intermediate waypoints instead of integrating a fine-grained path into the objective function.

The approach presented in [145] performs the planning in the 5-dimensional (x, y, θ, v, w) space. This allows to overcome the problem of a too high velocity before introducing a sharp turn. An A^*-search is supported by numerous heuristics that make the search problem feasible. One of the heuristics is to first search a path in the (x, y) plane that defines a corridor for the subsequent search in the 5-dimensional search space. Thus, it is not obvious, how one can efficiently circumvent the circular shape assumption without deeply interfering with the chosen heuristics. Nevertheless, the interesting idea of using the 5-dimensional (x, y, θ, v, w) space is that this planning system can also produce clothoidal trajectories that are naturally needed in many cases of exactly passing a goal point.

The approach presented in [4] is based on similar ideas as the *curvature distance lookup* method and the authors thus refer to the *CDL* approach. They do not use lookup tables and perform the distance calculation online. However, that can be done only as long as the polygonal contour is as simple as a rectangle. Even then, only a subset of laser scan points is used. However, their experiments further confirm the robustness of this class of approaches.

3.4 The *Curvature Distance Lookup* Approach

Motion execution algorithms use current sensor information to determine a motion command for the next time step. An approach that is particularly well suited for robots operating at high speed is the *dynamic window approach* since it takes into account kinematic and dynamic constraints. However, with reasonable computing power, for example, it is not able to process raw laser range scans and it thus requires preprocessing for data reduction. However, the far most limiting constraint is the assumption of a circular robot shape. The presented *curvature distance lookup* approach extends the dynamic window approach such that it can handle any-shaped robots while still being able to consider kinematic and dynamic constraints. Furthermore, the presented extension reduces the computational load such that one can even use raw laser range scans without any data reduction. This section first presents the dynamic window approach that is subsequently extended to the *curvature distance lookup* approach.

3.4.1 The Dynamic Window Approach

The *dynamic window approach* is able to handle kinematic and dynamic constraints but assumes a circular robot shape. The description is geared to [47] where further details can be found. The dynamic window approach is designed for mobile robots that are equipped with a synchro-drive. A motion command of a synchro-drive is a set of (v, w) values. The translational velocity v and the rotational velocity w can be set independently of each other. A synchro-drive can adopt to arbitrary headings only in case it is not moving. As soon as the translational velocity is non-zero, it follows a circular arc whose curvature is determined by the ratio of the rotational and the translational velocity.

The dynamic window approach assures safe operation at all times by selecting only such motion commands that allow the mobile platform to come to a stop before collision. The search for the next motion command is carried out in the space of velocities. The dynamic window is a discretized part of the (v, w)-plane that is spanned by all values of the translational velocity v and the rotational velocity w that are reachable from the current (v, w) setting within the next time step without exceeding the maximum acceleration and deceleration values. Each grid cell of the dynamic window represents a motion command (v, w) that results in a circular trajectory. The motion command is admissible only if the free space along the circular trajectory is sufficient to come to a stop. From the set of admissible velocities, an objective function selects the most appropriate combination of v and w. It can take into account the progress towards the goal and can prefere high translational velocities, for example.

For the selection of the next motion command, the dynamic window approach solely considers one time step. For deciding whether a (v, w) tuple is admissible, it considers the remaining distance along the resulting path. The available free space has to be sufficient to come to a stop without colliding with an obstacle independently of the required time. Repeatedly applying the dynamic window approach results in piecewise circular paths.

Figure 3.1 shows the velocity space and the dynamic window as introduced in [47]. The space of motion commands V_s of a synchro-drive is spanned by the maximum and minimum values of the translational velocity v and the rotational velocity w. The dynamic window V_a takes into account the limited accelerations and restricts V_s to those (v, w) tuples that can be reached within one time step. It is centered around the current settings of v_c and w_c. Let a_v and a_w denote the maximum translational and rotational accelerations.

$$V_a = \Big\{(v, w) \mid v \in [v_c - a_v \cdot \Delta t, v_c + a_v \cdot \Delta t],\ w \in [w_c - a_w \cdot \Delta t, w_c + a_w \cdot \Delta t]\Big\} \tag{3.1}$$

Each grid cell of the dynamic window represents a motion command (v, w) that defines a specific curvature. For each curvature, one has to calculate the collision free path length $d(c, \mathcal{O}, \mathcal{S})$ when driving with curvature c given the obstacles $\mathcal{O}$ and the shape $\mathcal{S}$ of the robot. In case of the dynamic window approach, the representation of the shape of the robot solely consists of the radius of its circular shape and the obstacles are represented by obstacle lines as shown in figure 3.2.

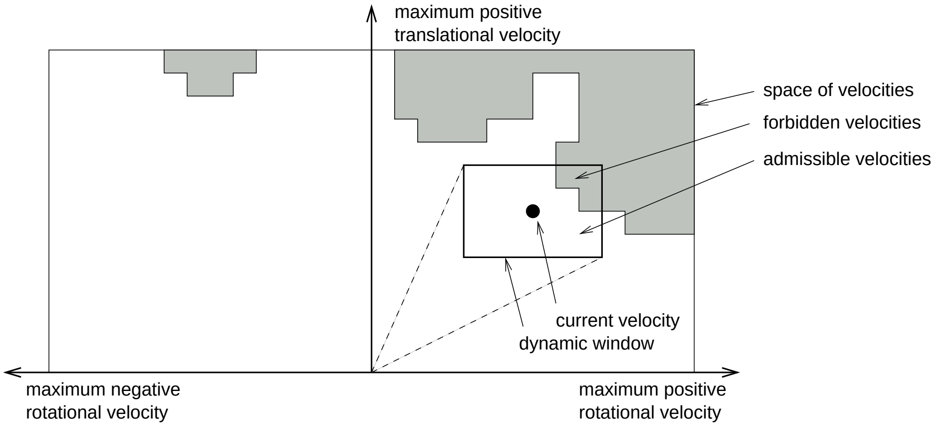


Figure 3.1: *The velocity space and the dynamic window as introduced in [47].*

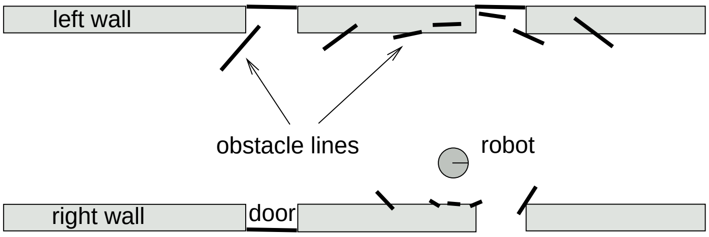


Figure 3.2: *Example environment of the* DWA *taken from [47].*

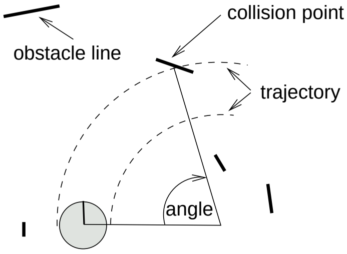


Figure 3.3: *Distance calculation of the* DWA *taken from [47].*

The distance calculation of the dynamic window approach is shown in figure 3.3. It calculates the points of intersection between all obstacle lines and both the inner and the outer trajectory. Then the minimum angle φ_{min} between the robot and all intersection points is determined. That allows to calculate the drivable arc length $d = \varphi_{min}(r_{inner} + r_{robot}) - r_{robot}$. The resulting arc length d can be converted into an angular distance $\gamma = d/(r_{inner} + r_{robot})$.

For each (v, w) tuple in the dynamic window V_a, it now has to be checked whether both the translational and the rotational component can achieve zero velocity on the remaining free path length. A (v, w) tuple is admissible only if both the translational and the rotational velocity are below the maximum allowed velocities regarding the remaining distance.

$$V_d = \left\{ (v, w) \mid v \leq \sqrt{2 \cdot d(c, \mathcal{O}, \mathcal{S}) \cdot a_{v,break}(c)} \ \wedge \ w \leq \sqrt{2 \cdot c \cdot d(c, \mathcal{O}, \mathcal{S}) \cdot a_{w,break}(c)} \right\} \tag{3.2}$$

Let $a_{v,break}(c)$ and $a_{w,break}(c)$ be the accelerations for breakage. It is important to note that $a_{v,break}$ and $a_{w,break}$ have to be chosen such that the brake maneuver takes place with the desired curvature. Depending on the maximum values for deceleration, either the translational or the rotational deceleration is the limiting part.

$$c = \frac{w}{v} \overset{!}{=} \frac{a_{w,break}}{a_{v,break}} \tag{3.3}$$

By the way, the dynamic window approach as described in [47] always solely considers the maximum values of breakage acceleration for $a_{v,break}$ and $a_{w,break}$. However, the maximum values define a curvature that is different from that of the considered (v, w) tuple. Thus, the remaining free space of the considered (v, w) tuple is checked against the curvature of the maximum decelerations that is different to the curvature of (v, w). It seems that this so far has not attracted attention since the rotational component typically provides very high accelerations compared to the translational component. Thus, in most of the cases, (v, w) tuples are rejected due to the translational component without having to consider the rotational component. However, it makes a big difference as soon as the translational velocity is very low. However, with a circular robot and a synchro-drive, a high rotational component with a very small translational command is like turning in place so that no collisions occur wih a circular shaped robot.

$$V_r = V_s \cap V_a \cap V_d \tag{3.4}$$

All (v, w) tuples that are in the set V_r cause no collision and can be reached within the next time step. Thus, it is safe to select any motion command from V_r. However, to achieve a goal oriented motion, one has to apply an objective function $G(v, w)$ that selects the most appropriate (v, w) tuple from V_r.

$$G(v,w) = \alpha \cdot heading(v,w) + \beta \cdot dist(v,w) + \gamma \cdot velocity(v,w) \tag{3.5}$$

$$heading(v,w) = 1 - \frac{\mid \Theta - w \cdot \Delta t \mid}{\pi} \tag{3.6}$$

$$dist(v,w) = \frac{d(c, \mathcal{O}, \mathcal{S})}{L} \tag{3.7}$$

$$velocity(v,w) = \frac{v}{v_{max}} \tag{3.8}$$

The different terms of the objective function are normalized to $[0, 1]$. The *heading* measures the alignment of the robot with the target direction. Θ is the goal heading relative to the robot and must not be confused with a heading at the goal. The *dist* term favours curvatures that provide a longer distance until collision. L is the maximum considered free path length. Finally, *velocity* prefers high translational velocities. Thus, the objective function $G(v, w)$ favours high speed, curvatures that provide a greater distance to collision and curvatures that head into the goal. The parameters α, β and γ weight the different terms. Typical values are $\alpha = 2.0$, $\beta = 0.2$ and $\gamma = 0.2$.

One can easily integrate further restrictions on V_r. Often, for example, it makes sense to adjust the velocities according to the side clearance of the robot which is denoted by V_n. Furthermore, one should forbid unsafe (v, w) tuples by introducing V_f. For example, it is not allowed to drive too fast through narrow curves. V_r is then given by $V_s \cap V_a \cap V_d \cap V_n \cap V_f$.

3.4.2 The Problem of Calculating the Distance to Collision

The time-consuming operation of the dynamic window approach is the calculation of the remaining distance along a circular path. Each (v, w) tuple results in a curvature and for each curvature inside the dynamic window, one has to determine the distance $d(c, \mathcal{O}, \mathcal{S})$. The distance d not only depends on the curvature c, but also on the shape of the robot and the obstacle configuration. Since the obstacle configuration changes constantly due to the movement of the platform and also due to the dynamics of the environment, one has to calculate d within the control loop of the dynamic window approach.

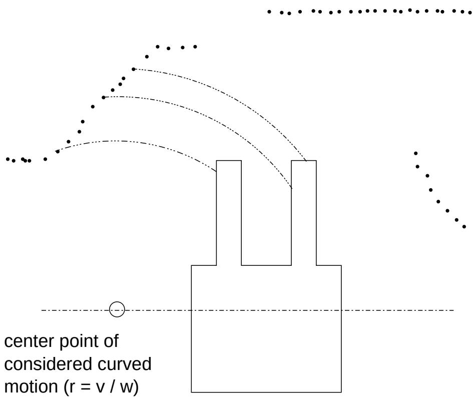


Figure 3.4: *The distance calculation in case of an any-shaped robot and high volume sensor data.*

Figure 3.4 shows an any-shaped robot and high volume sensor data as, for example, provided by a laser range finder. To calculate the minimum distance for one curvature, one needs to intersect all obstacles with the shape of the robot. In case of n_c curvatures and a polygonal shape with n_p segments and n_o obstacles, one gets complexity $O(n_c \cdot n_p \cdot n_o)$ which is $O(n^3)$.

The dynamic window approach reduces the complexity by assuming a circular shape. There, it is sufficient to solely consider the inner and the outer trajectory of the robot. Furthermore, one can easily reject those obstacles that do not lie in the annulus of the inner and outer trajectory. Even then, only obstacle lines are assumed to provide a significant data reduction compared to raw laser range scans.

3.4.3 The *Curvature Distance Lookup* Approach

The *curvature distance lookup* approach is the extension of the *dynamic window approach* to any-shaped robots. It uses lookup tables for the distance calculation and divides the distance calculation into an *offline* and an *online* part.

3.4.3.1 The Key towards an Any-Shaped Contour

A decisive role is played by two observations. The first observation is that the distance should be calculated in the robot frame instead of the world frame. Using the robot frame, obstacles at the same relative position contribute to the remaining distance in the same way independently of where this setting occured in the world frame. Of course, the effect of a relative obstacle position is different for each curvature. None the less, one can now precalculate the effect of any relative obstacle position on any curvature. By discretizing the local environment of the robot and the curvatures, one can use a lookup table to store those values. The discretization of the curvatures is already given by the discrete dynamic window. Thus, the lookup table $T_{distance}$ maps a relative cartesian position (i_x, i_y) and a curvature i_c to the remaining free distance d, that is $T_{distance}(\mathcal{S}) : (i_x, i_y, i_c) \mapsto (d)$.

The *offline* part is used to perform the time-consuming calculation of the intersections of the possible obstacle locations with the robot shape. The robot shape can now be of any complexity since that part of the calculation is removed from the control loop. The control loop solely executes the *online* part of the distance calculation. First, the indices (i_x, i_y) of all grid cells occupied by obstacles are determined. For a curvature c with the curvature index i_c, the distance $d(c, \mathcal{O}, \mathcal{S})$ is then the minimum over all distances $d_i = T_{distance}(i_x, i_y, i_c)$ of the occupied cells. The *online* part can be performed very fast.

The second observation is that the number of curvatures is only linear in the number of v and w values. That is crucial since otherwise the size of the lookup tables would not be reasonable anymore.

3.4.3.2 The Offline Calculation of the Lookup Tables

We now introduce the various lookup tables used by the *curvature distance lookup* approach. The $T_{distance}$ lookup table plays the key role of the *CDL* approach.

Curvature For each (i_v, i_w) tuple of V_s, one needs to know the curvature index i_c. The lookup table $T_{curvature}$ maps the index (i_v, i_w) of a velocity tuple of V_s onto the curvature index i_c.

$$T_{curvature} : (i_v, i_w) \mapsto (i_c) \tag{3.9}$$

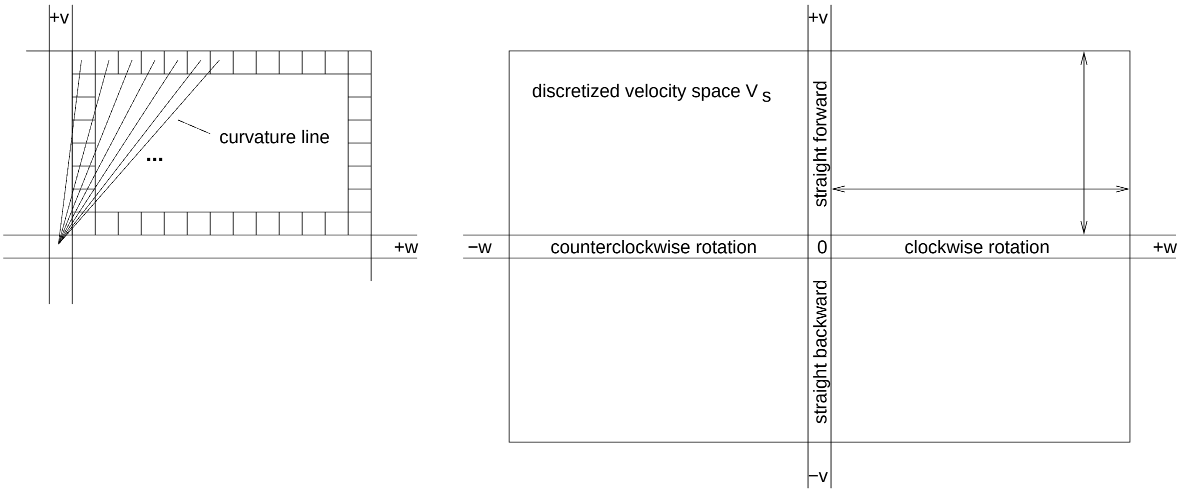


Figure 3.5: *The curvature lookup table for mapping velocity indices into a curvature index.*

Figure 3.5 shows the structure of the lookup table. Its size is determined by the discretization of the velocity space. The curvature is the reciprocal value of the gradient of a curvature line. It is sufficient to distinguish the curvature of the boundary cells only and to assign the inner cells that curvature index that represents the closest curvature value. There are five distinguished curvature indices listed in table 3.1.

Table 3.2 lists the typical values of a *RWI B21* platform. Excluding the distinguished curvature indices, the first quadrant has $(n_v - 1) + (n_w - 1) + (n_v - 1) + (n_w - 1)$ different curvatures with $n_v = \frac{v_{max}}{v_{step}}, n_w = \frac{w_{max}}{w_{step}}$.

Acceleration For each curvature c_i, one needs to know the maximum accelerations for breakage $(a_{v,break,c_i}, a_{w,break,c_i})$. It depends on the curvature whether the translational or the rotational acceleration is the limiting factor.

$$T_{accel} : (i_c) \mapsto (a_{v,break,c_i}, a_{w,break,c_i}) \tag{3.10}$$

$i_{c,0}$	No motion at all. The curvature and the radius is undefined.
$i_{c,+v}$	Motion straightforward with positive translational component and without any rotational component. The curvature is zero and the radius is undefined.
$i_{c,-v}$	Motion backwards with negative translational component and without any rotational component. The curvature is zero and the radius is undefined.
$i_{c,+w}$	Clockwise rotation in place. The curvature is undefined and the radius is zero.
$i_{c,-w}$	Counterclockwise rotation in place. The curvature is undefined and the radius is zero.

Table 3.1: *The five distinguished curvature indices.*

v_{min}	-1000 mm/s	v_{max}	+1000 mm/s	v_{step}	10 mm/s
w_{min}	-70 deg/s	w_{max}	+70 deg/s	w_{step}	1 deg/s

number of curvatures per quadrant	336
total number of curvatures	1349

Table 3.2: *A typical set of values of V_s.*

The size of this lookup table corresponds to the number of different curvatures. The content of the lookup table is determined according to equation 3.11.

$$\begin{array}{lll}
\text{if} \quad c = 0 & & \left\{\begin{array}{l} a_{v,break,c} = a_{v,break,max} \\ a_{w,break,c} = 0 \end{array}\right. \\
\text{else if} \quad c \to \infty & & \left\{\begin{array}{l} a_{v,break,c} = 0 \\ a_{w,break,c} = a_{w,break,max} \end{array}\right. \\
\text{else} & c \leq \frac{a_{w,break,max}}{a_{v,break,max}} & \left\{\begin{array}{l} a_{v,break,c} = a_{v,break,max} \\ a_{w,break,c} = c \cdot a_{v,break,max} \end{array}\right. \\
 & c > \frac{a_{w,break,max}}{a_{v,break,max}} & \left\{\begin{array}{l} a_{v,break,c} = \frac{a_{w,break,max}}{c} \\ a_{w,break,c} = a_{w,break,max} \end{array}\right.
\end{array} \tag{3.11}$$

Distance For each grid cell and each curvature, the distance lookup table stores the remaining distance until collision with the grid cell when driving with the respective curvature. It stores a tuple that contains the distance and the angle even though one can convert both by means of the curvature. However, due to the singularity of the curvature and due to numeric problems, it makes sense to always hold both values. Of course, either the distance or the angle value is invalid in case of the distinguished curvature indices.

$$T_{distance} : (i_x, i_y, i_c) \mapsto (d_{i_x,i_y,i_c}, \varphi_{i_x,i_y,i_c}) \tag{3.12}$$

The distance lookup table is shown in figure 3.6. Its size is determined by the discretization of the

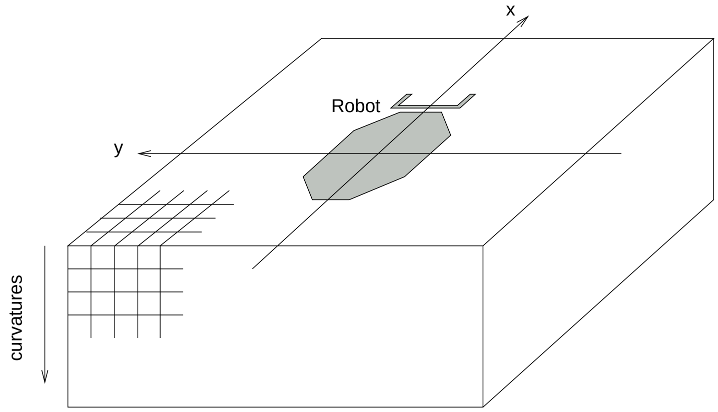


Figure 3.6: *The distance lookup table.*

cartesian space of the surroundings of the robot and by the number of curvatures. A typical setting uses grid cells of size 100 × 100 mm^2 to represent a ± 3000 × 3000 mm^2 section with the robot in the center. Thus, the representation of the cartesian work space consists of 3721 grid cells. Refering to the above example, the distance lookup table consists of 5 019 629 entries.

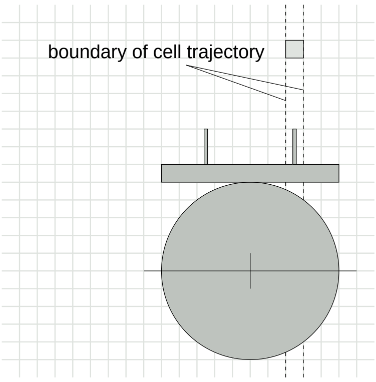


Figure 3.7: *The calculation of the entries of the distance lookup table in case of zero curvature.*

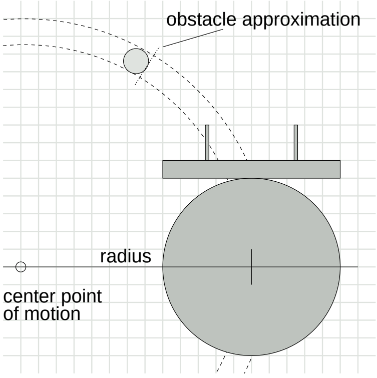


Figure 3.8: *The calculation of the entries of the distance lookup table in case of a regular curvature.*

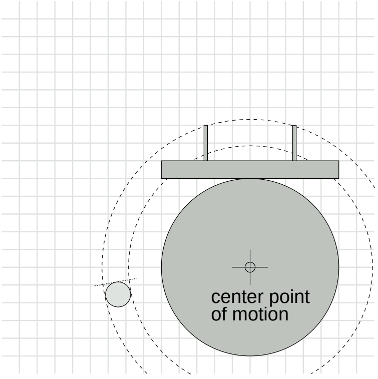


Figure 3.9: *The calculation of the entries of the distance lookup table in case of zero radius.*

The distance lookup table requires to calculate the free distance from the robot to any of the grid cells of the cartesian work space. That has to be done separately for each curvature in $T_{distance}$. Figure 3.7 shows the special case of a straight motion. The boundaries of the cell trajectory form straight lines. First, one has to calculate the intersection points $\mathcal{P}$ of those lines with all shape elements of the robot. Then, one has to find all vertices of the robot shape that lie within the boundaries of the cell trajectory and add these to $\mathcal{P}$. In case of a circular shaped robot part, one only considers the segment of the circle that lies within the boundaries of the cell trajectory. One then selects that point that gives the shortest distance to the occupied cell. Normally, however, the circular shaped parts are approximated by a polygon. Since that affects the complexity of the offline calculation only, it does not matter to introduce a high number of contour segments. Finally, the remaining free space for all points in $\mathcal{P}$ is calculated and the minimum value is stored in $T_{distance}$. In case of a straight motion, the angular distance is set to zero.

Figure 3.8 shows the distance calculation for a regular curved path. In order to simplify the calculations, the rectangular grid cell is enclosed by a circle. Each curvature defines a center point of motion that forms the center point of circles for the inner and the outer boundaries of the cell trajectory. In principle, the same calculations as before are performed, but now the relevant parts of the contour of the robot lie within an annulus. The distance calculation is performed with respect to the obstacle approximation. By means of the curvature, one can easily calculate the angular distance from the arc length and vice versa.

Figure 3.9 shows a motion with zero radius that is turning in place. Now, the center point of motion is equivalent to the center point of the vehicle. In principle, one again performs the same calculations but now only considers the angular distance. The distance values of those entries are set to zero in $T_{distance}$.

Since those calculations are performed offline, their complexity is relevant only in case one has to calculate a new lookup table. That is necessary in case one picks up a payload. However, one normally already holds several instances of the lookup tables to cover the most often needed shapes of the mobile platform so that one can switch between them at runtime.

Miscellaneous The lookup table T_{nice} is used by the constraint V_n to restrict the velocities depending on the obstacle configuration. For each grid cell of the cartesian work space of the surrounding of the robot, it provides maximum values for the translational and the rotational velocity. Thus, depending on the side clearance, for example, one can reduce the speed of the robot. It depends on the application how one sets these values. Even though the robot only selects safe motion commands, reducing the velocities in case there are obstacles nearby often looks less aggressive and more trustable which is an important factor towards acceptance.

$$T_{nice} : (i_x, i_y) \mapsto (v_{nice,i_x,i_y}, w_{nice,i_x,i_y}) \tag{3.13}$$

The lookup table $T_{forbidden}$ is used by the constraint V_f to reject all (v, w) tuples that result in an unsafe operation. For example, certain (v, w) tuples might cause the robot to topple down. Furthermore, driving narrow curves with high speed is dangerous. Due to the limited viewing angle of the laser range finder, for example, new obstacles can suddenly become visible so that there is no chance to react anymore.

$$T_{forbidden} : (i_v, i_w) \mapsto \{0, 1\} \tag{3.14}$$

3.4.3.3 Online Use of the Lookup Table

The online use of the lookup tables is summarized in figure 3.10. It shows the control loop of the *curvature distance lookup* method and illustrates how the various lookup tables are included into the selection of the motion command.

The current state of the robot defines the dynamic window ①. The not allowed (v, w) tuples inside the dynamic window are masked by means of the $T_{forbidden}$ lookup table ②. By means of the $T_{curvature}$ lookup table ③, all curvature indices belonging to allowed (v, w) tuples of the dynamic window are accumulated in the set of curvatures $\mathcal{C}$ ④.

All available obstacle information is mapped into the grid representation of the cartesian work space of the robot ⑤. The laser range scan is neither preprocessed nor is there any data reduction step.

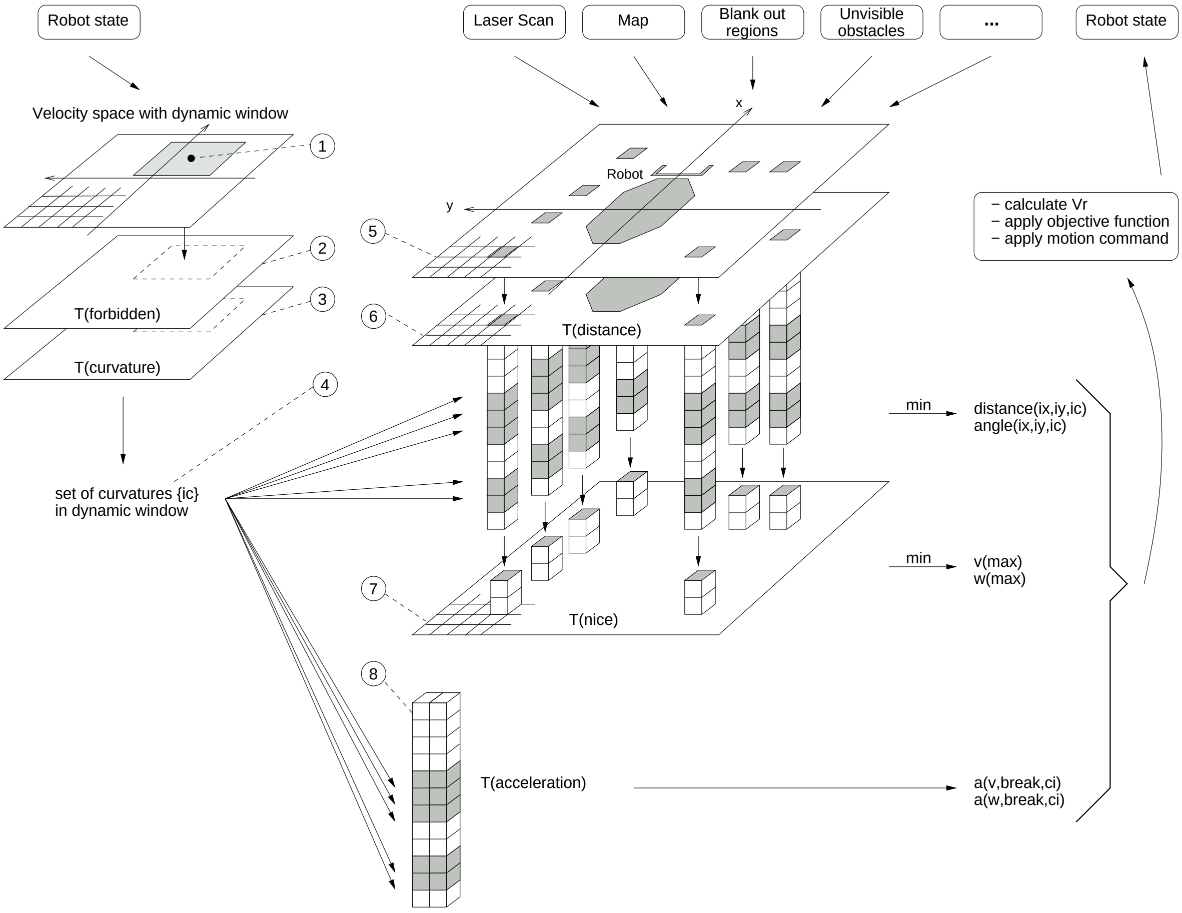


Figure 3.10: *The online use of the lookup tables.*

Typically, the laser range scan is the latest integrated obstacle information since it allows to overwrite outdated obstacle configurations. These are typically provided by longterm maps since these require several validations before they adapt to a new obstacle configuration. The result of this step is a list of indices $\mathcal{Q}$ that denote occupied grid cells.

The next step is to determine the remaining free space ⑥. Both $\mathcal{C}$ and $\mathcal{Q}$ only contain a small subset of the set of curvature indices and the set of obstacle grid cells, respectively. Thus, the following step can be performed very fast even if it has to be executed for each curvature index $i_c \in \mathcal{C}$.

$$d(i_c) = \min_{(i_x,i_y)\in\mathcal{Q}} d(i_x,i_y,i_c) \tag{3.15}$$

$$\varphi(i_c) = \min_{(i_x,i_y)\in\mathcal{Q}} \varphi(i_x,i_y,i_c) \tag{3.16}$$

$$\tag{3.17}$$

Finally, one has to determine the maximum velocities accepted as nice operation of the robot ⑦.

$$v_{nice,max} = \min_{(i_x,i_y)\in\mathcal{Q}} v_{nice}(i_x,i_y) \tag{3.18}$$

$$w_{nice,max} = \min_{(i_x,i_y)\in Q} w_{nice}(i_x,i_y) \tag{3.19}$$

Now, everything needed to calculate V_r is available. The maximum accelerations for breakage ⑧ are looked up with calculating V_d.

$$\begin{aligned}
V_d &= \left\{(v_i,w_i) \mid v_i \le \sqrt{2\cdot d(i_c)\cdot a_{v,break}(i_c)} \ \wedge\ w_i \le \sqrt{2\cdot \varphi(i_c)\cdot a_{w,break}(i_c)}\ \right\} && (3.20)\\
V_n &= \left\{(v_i,w_i) \mid v_i \le\ v_{nice,max}\ \wedge\ w_i \le\ w_{nice,max}\right\} && (3.21)\\
V_f &= \left\{(v_i,w_i) \mid (v_i,w_i) \quad \text{allowed by} \quad T_{forbidden}\right\} && (3.22)\\
& && (3.23)\\
V_r &= V_s\cap V_a\cap V_d\cap V_n\cap V_f && (3.24)
\end{aligned}$$

3.4.3.4 Motion Control Strategies

The standard objective function in form of equation 3.5 is designed to move the robot towards the goal with high velocity. It neither considers stopping at the goal point nor the heading of the robot at the goal point.

Stopping at a Goal In case the robot wants to stop at the goal, it has to favor such velocities that allow the robot to come to a stop at the goal while still approaching the goal with a reasonable speed. Thus, the velocity term uses a different evaluation function as soon as the euclidean distance $d_2(x_r,y_r,x_g,y_g)$ between the robot and the goal is below a threshold d_{close}.

$$velocity(v,w) = \begin{cases} \frac{v}{v_{max}} & \text{if} \quad d_2 > d_{close} \\ \frac{v}{v_{target}} & \text{if} \quad d_2 \le d_{close}\ \wedge\ v < v_{target} \\ 0 & \text{if} \quad d_2 \le d_{close}\ \wedge\ v \ge v_{target} \end{cases} \tag{3.25}$$

$$v_{target} = \sqrt{2\cdot d_2(x_r,y_r,x_g,y_g)\cdot\kappa\cdot a_{v,break,max}} \tag{3.26}$$

The target velocity v_{target} is the maximum velocity that allows the robot to stop on the distance to the goal. Since the distance is approximated by the euclidean distance, it is either correct or optimistic with respect to the path length executed by the robot. Furthermore, the damping factor κ reduces the maximum acceleration $a_{v,break,max}$ for calculating v_{target}. Thus, the goal is approached without relying on the maximum deceleration. The objective function favors (v,w) tuples with a translational component that is close to v_{target} and ignores (v,w) tuples with a translational velocity above v_{target}. The discontinuity in the evaluation function when exceeding v_{target} does not cause any troubles since the evaluation function only selects among admissible (v,w) tuples. Thus, exceeding v_{target} due to a translational velocity near the discontinuity in the evaluation function is not a safety issue and would only result in overshooting the goal region. However, since the calculation of v_{target} is based on a damped maximum acceleration, v_{target} is always a conservative value such that the robot comes to a stop for sure.

The above objective function tries to approach the goal on a straight line in such a way that the robot comes to a stop at the goal point. Of course, a goal *region* is used to decide whether the goal has been reached. The configuration of the size and shape of the goal region depends on the motion task that is executed. Mainly, a circular region is used. The objective function does not take into account the heading of the robot at the goal. In case of a synchro-drive robot, one can adjust the heading of the robot by turning in place provided that the current obstacle configuration allows to do so. The *CDL* approach correctly handles the case of turning in place.

Passing a Goal Passing a goal point is more pretentious than it appears in the first place. A typical setting is shown in figure 3.11. The robot moves along the hallway with a high translational velocity since the goal is still far away. Due to the dynamic constraints, it is then not able to make the sharp turn and thus misses the turn-off. Even worser is the lateral displacement shown in figure 3.12.

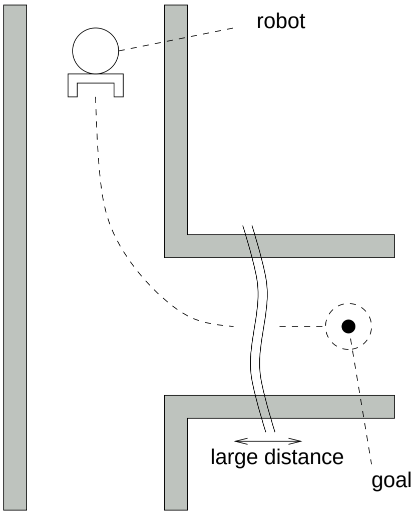


Figure 3.11: *Making a sharp turn.*

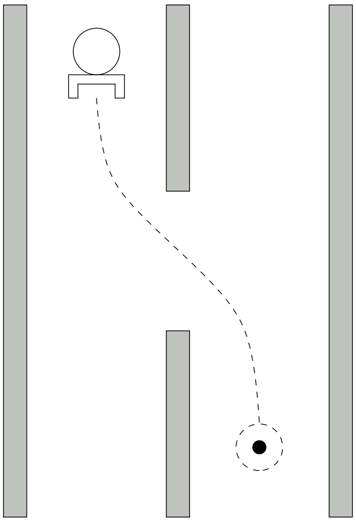


Figure 3.12: *Goal with lateral displacement.*

However, one can cope with such settings much better if one knows when one has to expect a sharp turn. It is sufficient to place an intermediate waypoint $wp(x_w, y_w, \theta_w)$ at the turn-off as shown in figure 3.13. One can now reduce the translational velocity depending on the relation between the robot and the intermediate waypoint wp. As soon as the euclidean distance of the robot to wp is below a threshold d_{slow}, one calculates $\alpha = \mid \alpha_1 \mid + \mid \alpha_2 \mid$ (α_1 equals to Θ in equation 3.6). If α is above a threshold α_{slow}, one sets v_{target} to reduce the translational velocity. The generation of such waypoints is described in section 3.5.1.

$$velocity(v, w) = \begin{cases} \frac{v}{v_{max}} & \text{if} \quad d_2 > d_{slow} \\ \frac{v}{v_{max}} & \text{if} \quad d_2 \leq d_{slow} \ \wedge \ \alpha \leq \alpha_{slow} \\ \frac{v}{v_{target}} & \text{if} \quad d_2 \leq d_{slow} \ \wedge \ \alpha > \alpha_{slow} \ \wedge \ v < v_{target} \\ 0 & \text{if} \quad d_2 \leq d_{slow} \ \wedge \ \alpha > \alpha_{slow} \ \wedge \ v \geq v_{target} \end{cases} \tag{3.27}$$

$$v_{target} = \max(v_{slow,min}, \sqrt{2 \cdot d_2(x_r, y_r, x_g, y_g) \cdot \kappa \cdot a_{v,break,max}}) \tag{3.28}$$

The effect of this heuristic is to slow down to a minimum speed $v_{slow,min}$ as soon as wp is approached but only in case the angular deviation is too big. As soon as the translational velocity is reduced, the objective function is able to adjust its heading towards wp. As soon as the robot is close enough to wp, activating the original goal point turns the robot into the desired direction. The minimum velocity $v_{slow,min}$ keeps the robot moving but is low enough that the robot can make a sharp

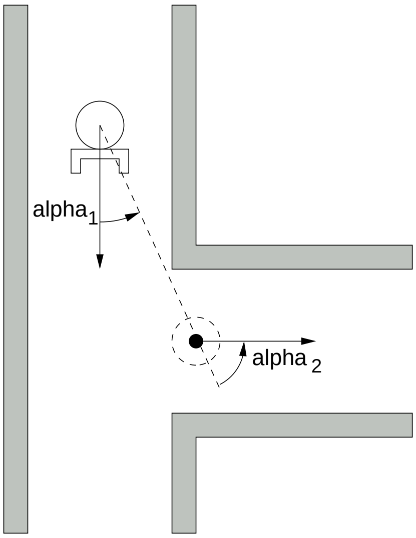


Figure 3.13: *Heuristic to approach intermediate waypoint.*

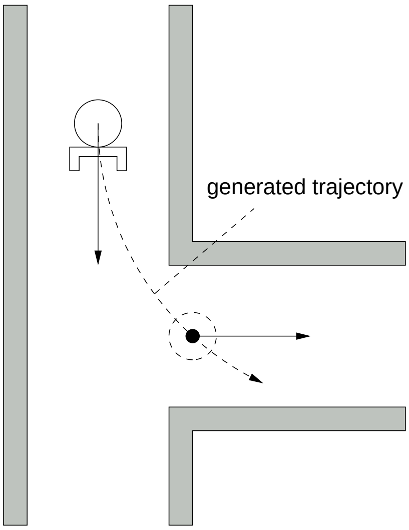


Figure 3.14: *Approaching intermediate waypoint.*

turn even in a narrow environment without requiring such high rotational velocities that all curvatures towards the goal point are forbidden. With the *RWI B21* platform, $v_{slow,min}$ is set to 150 mm/s. It is important to note, that $v_{slow,min}$ is only the *preferred* minimum translational velocity. The objective function always operates on the set of motion commands that never cause a collision. The robot is not slowed down and it even accelerates again if its angular deviation is below the threshold since in those situations, the robot has a chance to adapt to the desired heading even with a high translational velocity.

Figure 3.15 shows some typical settings. The robot does not slow down in case the angular deviation is low ①. In ②, the same setting but with a different heading at the goal is shown. Since the angular difference now is too large, the robot is slowed down. In ③, it is assumed that the robot is not able to head towards wp for some reasons. Thus, as soon as the angular difference is above the threshold, it again gets slowed down.

With this heuristic, the robot prefers a straight motion towards wp. It slows down the robot if its angular deviation expressed by α is too high. One has to recognize that this heuristic does not approach wp with the desired heading but slows down the robot such that the robot has a chance to adapt to the desired heading once it passed wp. The typical behavior is illustrated in figure 3.16. The robot tries to approach wp by a straight motion which results in a wall-following as soon as it is close to the wall. Due to its angular deviation, it is slowed down until it drives with $v_{slow,min}$ when it reaches wp. In case of an intermediate waypoint, one now switches to the next waypoint. Normally, θ_w is just determined such that it points towards the next goal point. Due to its low translational velocity, the robot is now able to turn into the desired direction. Thereby, it circumvents the corner of the wall very tightly.

Of course, this heuristic decelerates the robot to unnecessarily low translational velocities in case of sharp turns. Due to the limited viewing angle of the laser range finder, for example, sharp turns are typically situations in which not yet seen obstacles come into the field of view. Thus, it is a good idea to slow down if one has to expect not yet perceived obstacles that might even be very close.

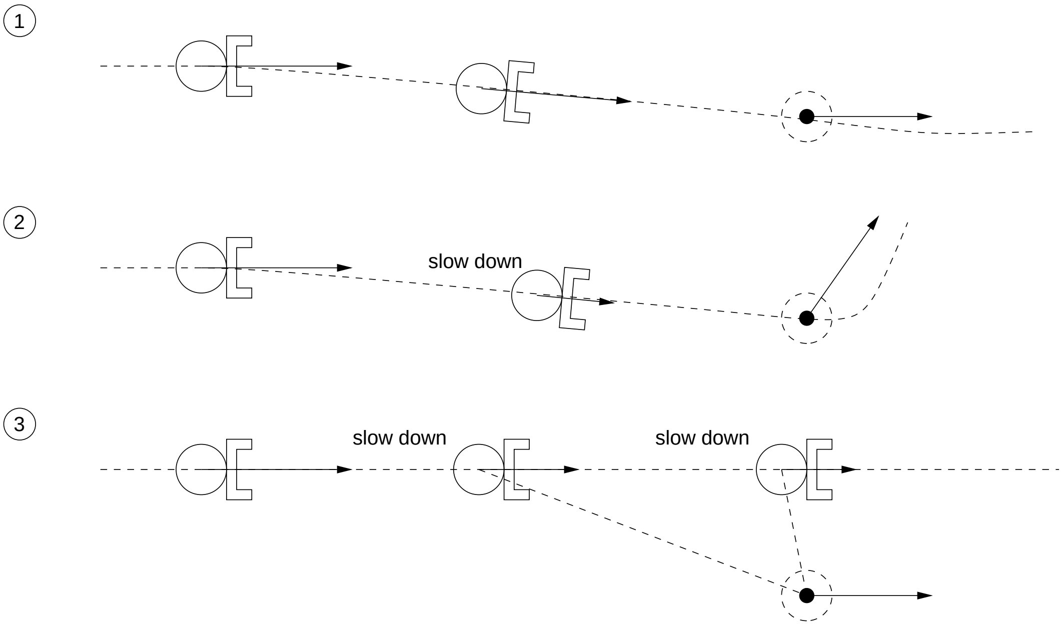


Figure 3.15: Slowing down due to angular deviations.

Using a Circular Trajectory There are many cases where tightly moving along corners with an any-shaped robot is not a feasible approach since one can get stucked. Such a situation is often encountered with narrow doorways, for example. Under these circumstances, it is much better to approach the goal on a circular trajectory as shown in figure 3.17. The circular trajectory can be used either to stop at a goal point with a desired heading or to pass a waypoint with a desired heading. As soon as the robot comes close to the waypoint wp ①, one tracks a point t ③ that moves along a circle c towards wp ①. In case wp is the goal point, t further follows the circle and in case of passing wp, t follows ④. The tracking is aborted as soon as the robot reached or passed wp. The circle c tangential to ① and through ② and the tracking point t on the circular arc respectively on the prolongation ④ are calculated within each cycle of the control loop. A typical value of the distance d_t between the robot and t is 500 mm. Following the moving tracking point results in a *dog curve*. The accuracy of reaching ① depends on the distance d_t. Of course, that can be considered by taking t from a larger concentric circle cc such that the robot follows c when t is moved on cc.

Again, one has to remind that t is only tried to be approached by the objective function. In case t lies in or behind obstacles, the objective function keeps the robot moving along the obstacle until it can head into the direction of t. As shown in figure 3.18, the robot then simply follows the wall. Of course, that does not work in all settings but it is a good heuristic in many of the encountered situations. To approach d_t, the above objective functions are applied and thus, the translational velocity foremost depends on d_t so that the robot drives with a reasonable low translational velocity.

It is important to note that in case of passing the waypoint, the effect of d_t if used as shown in figure 3.17 is a smoothing of the curvature near the discontinuity at ①. That smoothing already results in nice tracking points that can be approached smoothly by the objective function. However, d_t cannot be enlarged arbitrarily since then the robot deviates too much from the circular trajectory. Thus, the distance that can be used to smooth the curvature discontinuity is limited and thus, the maximum translational velocity is also restricted. In case the robot drives too fast, there is no way to fast enough

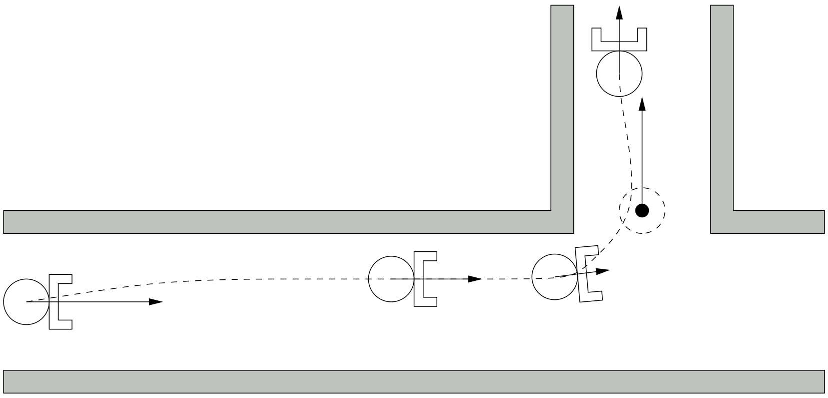


Figure 3.16: *Making a sharp turn into another hallway.*

bring down the rotational component and the robot overshoots the desired heading. Again, due to the dynamic window, the robot of course always only selects such motion commands that are safe. Thus, when overshooting, it is already slowed down such that it does not collide with any walls in the example settings. In principle, an optimized velocity profile for the above situation would require to consider clothoids, for example, if one wants to pass the waypoint with a nonzero translational velocity.

3.4.3.5 Other Kinematics

The approach can also be applied to kinematically equivalent configurations like a differential drive. For the differential drive, further constraints in the (v, w) space apply that are easily incorporated. Figure 3.19 shows an early implementation for a tricycle kinematic. Instead of the (v, w) search space, the (v, ϕ) space was used with v the translational velocity and ϕ the steering angle.

3.5 The Path Planning Component

The trajectory generation of the *curvature distance lookup* approach suffers from local minima due to the purely local decision on the next motion command. However, in many cases, it is nevertheless able to successfully reach a goal region without further support from a global component. Most important, the *CDL* approach always ensures a safe operation even in an unknown environment. Thus, deviating from a goal specification might result in getting stucked but not in a collision. For many circumstances, it is sufficient to provide suitable intermediate waypoints that are then approached by a trajectory generated by the *CDL* approach. It is even better to provide only few intermediate waypoints since that gives the *CDL* approach most freedom on how to approach them.

3.5.1 Path Planning at the Geometric Level

The *geometric path planning* component is an independent component that can be configured to interact with both the *map building* component and the *motion execution* component. Further details of these interactions are described in section 3.6. The *geometric path planning* component gets the so-called *current map* $M_c^{x_w,y_w,x_s,y_s,c_s,o_s}$ from the *map building* component. It is a grid map with origin (x_w, y_w), size (x_s, y_s), cell size c_s and obstacle growing o_s. It is always aligned to both the world

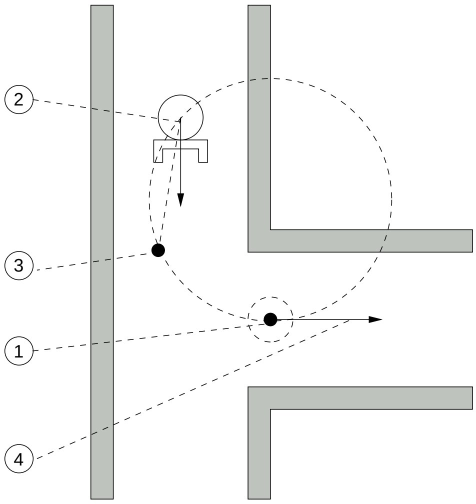


Figure 3.17: *Using a circular trajectory.*

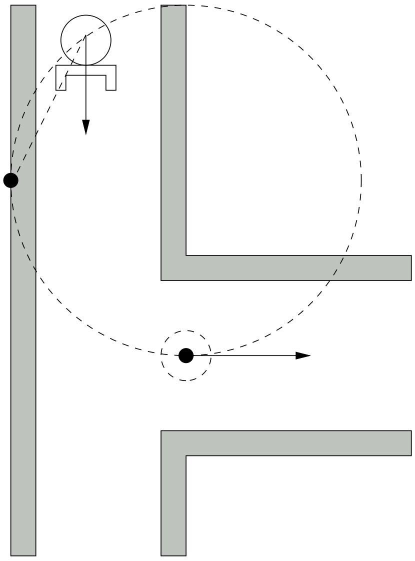


Figure 3.18: *The tracking point in or behind an obstacle.*

coordinate frame and the cells of the global grid map. The resolution must always be such that no cells of the global grid map are divided into sections when they are forwarded to the *current map*. In particular, the *current map* is neither rotated nor moved with the robot. Its offset, size and resolution is configured by the *sequencing layer* in connection with the configuration of the overall motion task. Its parameters are adjusted by the *sequencing layer* in accordance with the task execution progress like reaching another topological section.

Figure 3.20 shows the wave propagation algorithm that works on the map M_c. The wave propagation is started from the cells marked as goal cells. Goal cells do not have to be adjacent so that the robot can approach the closest goal region among different alternatives. One can either use the illustrated wave propagation that even does not store distance values in the grid cells or any other NF1 technique [99]. Obstacles are grown by an optimistic obstacle growing since the motion execution component with the *CDL* approach takes care to keep distance to obstacles.

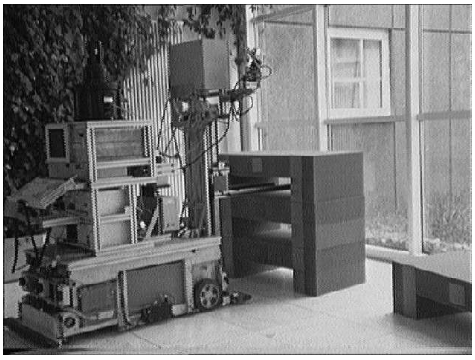


Figure 3.19: *A tricycle kinematic.*

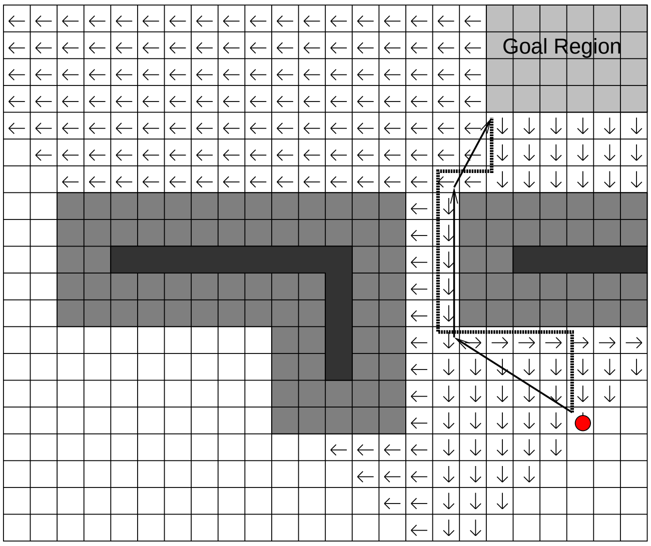


Figure 3.20: Path planning by wave propagation.

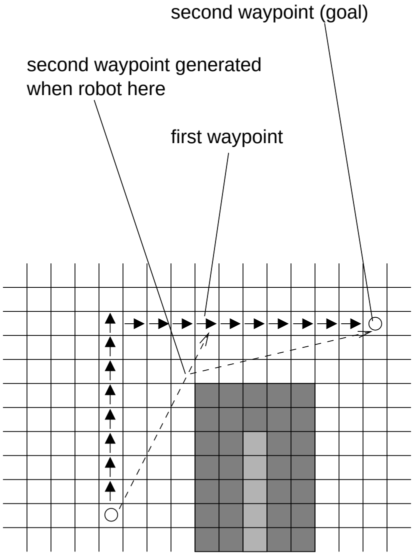


Figure 3.21: Effect of the shortening heuristic.

The important part is the *path shortening* heuristic illustrated in figure 3.21. It follows the planned path starting at the current robot position until the straight line shortening interferes with an obstacle. The prior cell on the path defines the next waypoint wp_1. The heading of wp_1 directs to wp_2. The waypoint wp_2 is obtained by applying the shortening heuristic a second time but now starting at the waypoint wp_1. In case wp_1 is a goal point, the heading depends on the goal point specification.

Since the geometric path planning is applied continuously, it generates a new waypoint as soon as one can further shorten the path. Thus, the resulting overall path is often close to the L_2 optimum path. Of course, a new waypoint is also obtained in case the path changed either due to a changed obstacle configuration or in case the robot had to avoid obstacles such that there is a better path to the goal region from its current position.

3.5.2 Path Planning at the Topological Level

The *topological path planning* component is located at the *deliberative layer*. Figure 3.22 shows a section of a topological map. The vertices A, B and C represent offices and the vertices G, H and I different hallways. The hallway H is divided into different sections, namely H_1, H_2 and H_3. The solid edges are enriched by the parameters for configuring the *current map* $M_c^{x_w,y_w,x_o,y_o,c_o,o_s}$ and the goal region. The dashed edges solely contain distance information to be able to correctly estimate the distances. The distance from A to B, for example, is estimated by $A - H_1 - H_2 - B$ and that from A to C by $A - H_1 - H_2 - H_3 - C$. With respect to executing the topological path, the dashed edges and the subsections H_x of H are of no relevance and then only H is considered as shown on the right of figure 3.22.

In the example, the grey shaded map patch belongs to the *bold* edge. Thus, when driving from B to H, the map patch is set such that the office B is contained in the map patch as well as that part of the hallway that serves as goal region. When driving from H to G, the map patch is adjusted to contain the hallway and the part of G that serves as goal region.

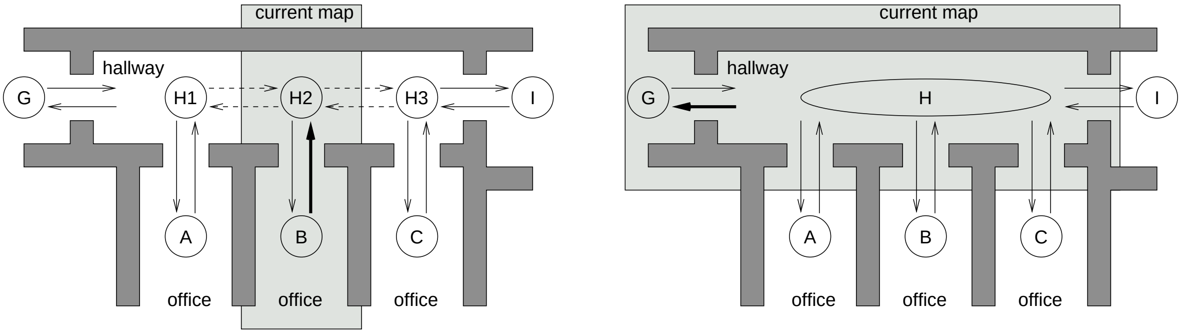


Figure 3.22: *Path planning at the topological level.*

3.6 The Integration

3.6.1 Approaching Maneuvers and the Robot Shape

Approaching maneuvers are particularly challenging with respect to collision avoidance in a setting as shown in figure 3.23. The robot approaches the table with its deployed manipulator to pick up the disc. In case one takes into account the manipulator, one cannot approach the table since the manipulator collides with the table. In case one ignores the manipulator, one takes the risk that the manipulator collides with an object in the environment while it moves towards the table. Thus, the *motion execution* component allows to configure *two* shapes that are evaluated both within the control loop. For each shape, one can specify polygonal shaped masks $\mathcal{D}$ that describe regions that are not considered for collision avoidance. These can be either specified relative to the robot coordinate system $\mathcal{D}_r$ or in the world coordinate frame $\mathcal{D}_w$. Furthermore, one can specify obstacles that have to be considered but that cannot be perceived by the sensors. These are also defined by closed polygons either relative to the robot $\mathcal{A}_r$ or in the world coordinate frame $\mathcal{A}_w$.

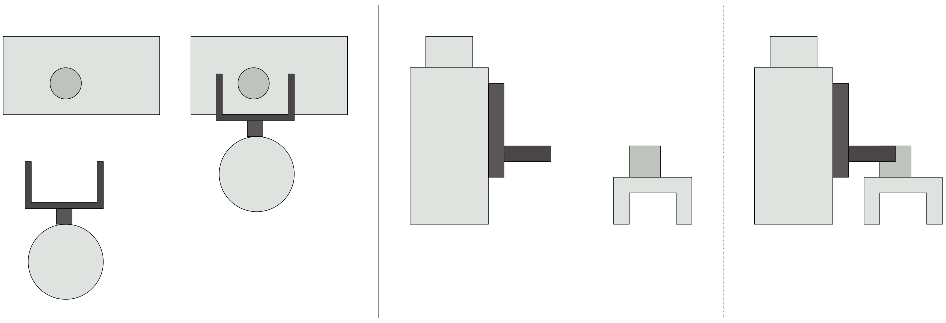


Figure 3.23: *Approaching maneuvers and the robot shape.*

In the example in figure 3.23, the first shape $\mathcal{S}_1$ solely contains the circular contour of the base of the robot. The second shape $\mathcal{S}_2$ contains the full shape of the robot including the manipulator. For it, a mask $\mathcal{D}_w$ in the world coordinate frame is defined to mask out the table and the disc is added as obstacle in the world coordinate frame $\mathcal{A}_w$ since it, for example, cannot be perceived as obstacle while approaching the table. The pose of both the table and the disc are sensed prior to invoking the approaching maneuver so that the masks correspond to the current pose of these objects. Outside the table, the complete shape of the robot is taken into account so that the manipulator cannot collide with an obstacle. It can even not collide with the disc. However, the table can also be approached without causing any harm since the shape of the enclosure prevents the robot from colliding with the table. As remark, approaching the disc with the manipulator is not possible with approximating the shape by a circle.

The mask $\mathcal{D}_r$ is used in case the manipulator is in the field of view of the laser range finder. That is needed with some pick-and-place operations. However, the robot then is driving almost blindly so that these kind of maneuvers are very unsafe.

3.6.2 Interaction of the Motion Execution and the Geometric Path Planning

The geometric path planning component and the motion execution component are loosely coupled. The motion execution always holds only one waypoint wp that can be overwritten on-the-fly by the geometric path planning component. In particular, there is no fine-grained path forwarded to the motion execution that has to be followed as close as possible. That allows the motion execution to apply its objective function over longer periods of time without encountering a change of the waypoint. Furthermore, that relieves the interaction between both components from complicated mechanisms to synchronize the current robot position with the path specification in case of updates and deviations.

The motion execution instantaneously reacts to a new goal configuration without violating any vehicle constraints. As long as the robot is far away from a waypoint, it tries to approach the waypoint by directly heading towards it which complies with the shortening heuristic. Thus, there are good chances to be able to successfully apply that strategy. In case there are any changes, the motion execution component anyway gets an update of its waypoint. As soon as the robot comes close to a waypoint, the further behavior depends on the selected objective function. Due to the optimistic obstacle growing, the intermediate waypoint wp might be too close to an obstacle. However, wp need not to be hit exactly and it is sufficient to enter a circular region with wp at its center.

Due to the shortening heuristic, intermediate waypoints are normally placed at such locations where something relevant with respect to motion execution happens. Thus, an intermediate waypoint carries the information that in case of being far away from it, one does, for example, not have to expect any sharp turns. An intermediate waypoint provides just that information that is needed to slow down to not miss a door or a turnout. Due to that characteristic of the intermediate waypoints, they can be used as basis for refined motion control strategies as described in section 3.4.3.4. In contrast thereto, with approaches that tightly integrate a navigation function and that follow the gradient at the fine-grained level of the cell size, one does not have any outstanding waypoints. Integrating an appropriate look-ahead is not easy since the fine-grained path makes only sense in case it considers the vehicle kinematics, for example. Then, however, it is already smoothed in some sense and finding critical sections is not an easy task. Resigning the intermediate waypoints results in the well-known drawbacks of the objective function that prefers high velocities such that sharp turns are missed.

3.6.3 Interaction with the Global Path Planner

In case of passing a waypoint wp, the motion execution switches to an objective function that favors driving into the direction of the heading of wp until an updated waypoint is received. In case of reaching a goal point, the *sequencing layer* is notified about that event. In case the goal is not already the final goal but is only an intermediate goal on the topological path, the motion execution was configured such that it continues to drive into the direction of the heading of the last goal point. In case no heading is defined, the robot just tries to move straightforward, always monitored by the *CDL* approach. Meanwhile, the sequencing layer gets the parameters of the next path segment from the topological map and performs the reconfiguration of the components. Since that is done very fast, the robot typically does not move more than 100 mm until it gets the next valid waypoint. In this way, the robot switches to the next path segment of the topological plan without stopping.

Of course, due to the simplified path planning approach, the robot can get stucked but that is detected and reported to the *sequencing layer* that is responsible for configuring the overall navigation task.

3.6.4 Survey on the Involved Components

Figure 3.24 shows the involved components in case of a typical navigation task. The configuration and the wiring of the components as well as the monitoring of the execution progress is done by the *sequencing layer*. Other configurations, for example, connect a vision based object tracking component with the motion execution component so that the robot tracks an object without colliding with obstacles. All experiments were performed with a translational speed of up to 1 m/s. The *laser server* provided 8 $scans/s$ with each consisting of 361 distance measurements and a field of view of 180 degrees. All shown components are run on a single computer (dual Pentium Pro 200 MHz) on-board the robot without using it to full capacity.

3.7 Experiments and Results

3.7.1 Moving with Deployed Manipulator

Figure 3.25 shows the *RWI B21* robot moving amongst stationary obstacles with a deployed manipulator. The trajectory is generated by the motion execution component without the geometric path planner. The shape of the robot is correctly modelled including the deployed manipulator. The overall path is approximately 7 m and the robot almost continuously drives with 800 mm/s. The obstacles are perceived during the motion by means of the laser range finder and are not known previously.

Figure 3.26 shows two persons blocking the path of the robot. In the following, images are referred to by *(row, column)*. In image (2/3), the robot turns to the right to avoid a collision. As can be seen by the heading of the robot, the persons jumped in front of the robot such that it could just avoid a collision. In image (3/1), one can see how close the robot passes the person. With a circular shape approximation, the robot would have to keep a much greater distance.

3.7.2 Moving in Cafeteria of FAW

Figure 3.27 shows the *RWI B21* platform driving through the cafeteria of *FAW*. Again, the environment is completely unknown to the motion execution component. Solely the self-localization component has a model in form of laser range scans to continuously adjust the pose of the robot. The path specification consists of two points, the first approximately at the position of the robot in image (3/3) and the second in that part of the cafeteria visible in image (5/3). Again, no geometric path planner is used and the trajectory is generated reactively by the motion execution component. The narrow passage between the pillar and the staircase is shown in image (1/1). The robot only slightly slows down at the narrow passage and again moves with approximately 800 mm/s in the other parts of the cafeteria.

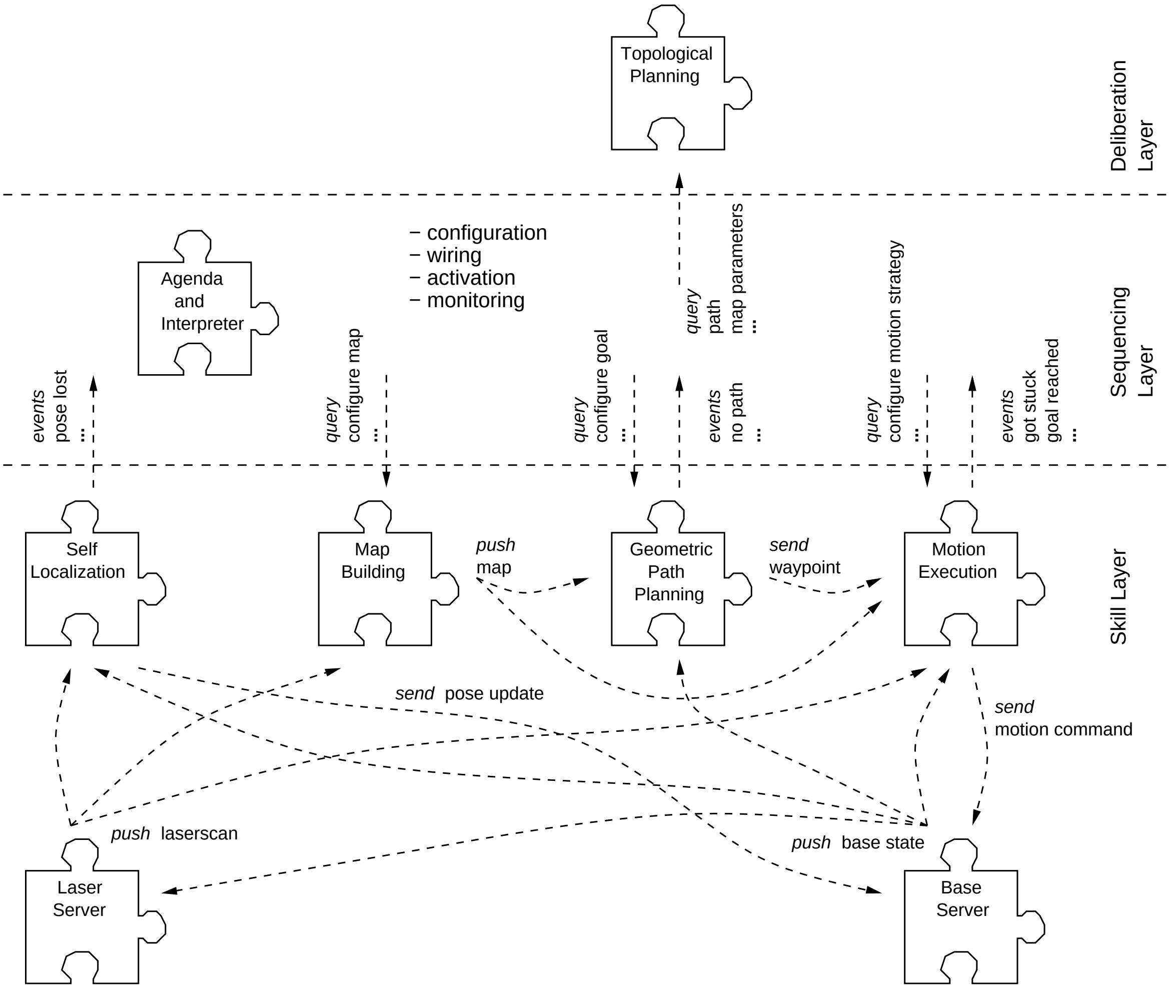


Figure 3.24: *A typical wiring of the components for a navigation task.*

3.7.3 Motion Control under Dynamic Constraints

Again, solely the motion execution component is used without any geometric path planning and without any prior knowledge about the environment. In all examples, the robot accelerates to approximately 900 mm/s without significantly slowing down during the movements. The specification of the path consists of two waypoints, the first in the section behind the obstacles and the second at the starting point of the experiments.

Figure 3.28 shows the robot passing the obstacles and then returning towards the starting point. The robot takes the opportunity of the opened passage to approach the starting point on the shortest path. The robot perceives the obstacle configuration by means of the laser range finder as set of distances. It then reacts to the set of obstacle points without extracting any higher level descriptions like the width of a passage, for example.

In figure 3.29 the same motion task is executed. It drives towards the first waypoint on the shortest path. When the robot already heads towards the passage, it is closed to force the robot to circumvent the obstacles. Due to the high translational velocity and plenty of free space, the robot passes the

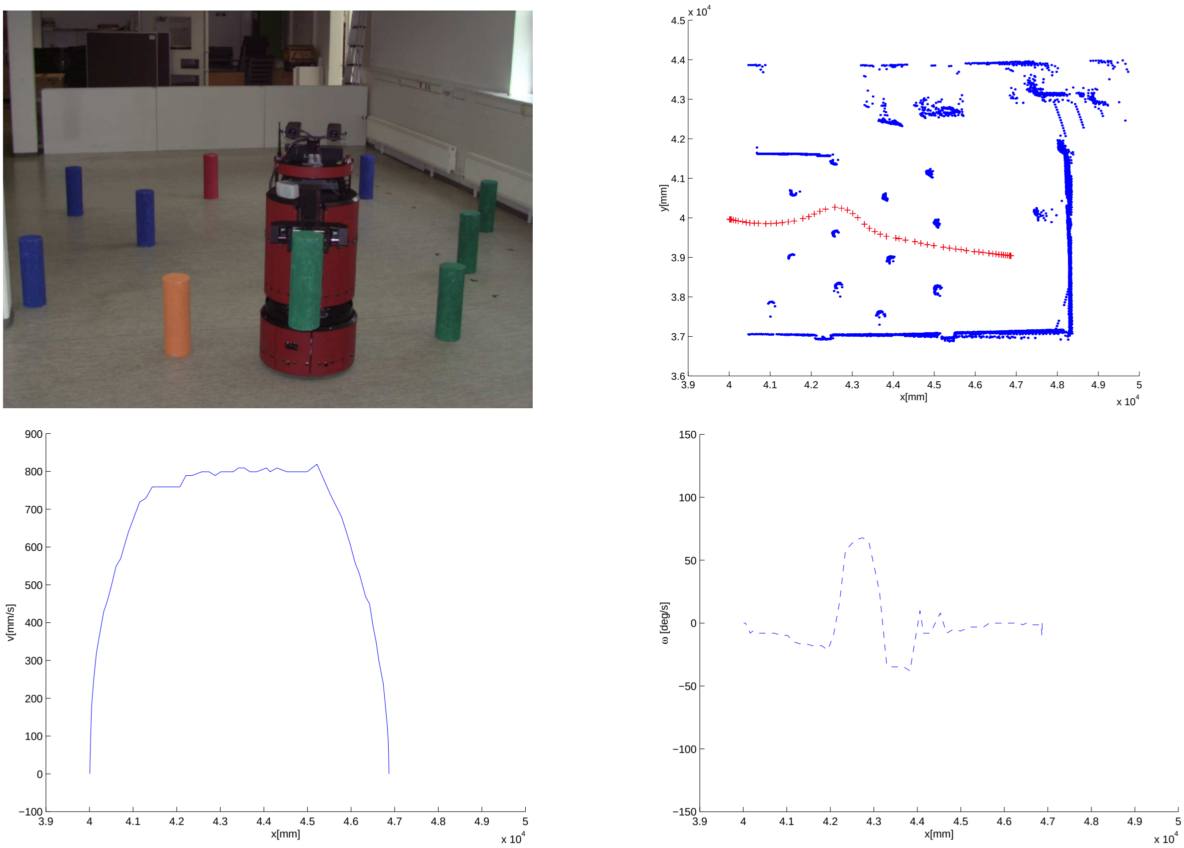


Figure 3.25: *Moving with deployed manipulator amongst stationary obstacles.*

obstacles in a big bend before it heads to the starting point.

In figure 3.30, the robot is started with an offset to the right. Thus, the shortest path towards the first waypoint is to drive in-between the obstacles. Opening the passage causes the robot to take the opportunity to reach the waypoint on the shortest path. As soon as the robot passed the passage, it is closed again. Thus, when the robot returns, it heads towards the free space besides the obstacles. However, the obstacle configuration is changed just while the robot is already heading towards the previously free section and the robot makes a turn to avoid a collision. Since it drove towards the obstacles with maximum velocity due to the previously free space, the turn is achieved with a very high rotational velocity. Thus, due to its dynamic constraints, it is not able to turn right to take the new opening. Thus, it drives a loop and takes the new opening then.

3.7.4 Moving with High Speed in the Robotics Evaluation Area

All motion tasks in the robotics evaluation area at the basement of *FAW* are performed with using the overall navigation system including self-localization, geometric and topological path planning and map building. Figure 3.31 shows a sequence taken from a motion along the hallway towards room three. In the hallway, the robot accelerates to 900 mm/s and passes the first room to its right with high speed as shown in image (3/3). In sufficient time before approaching room three, the motion execution component slows down due to the encountered angular deviation of the heading of the intermediate

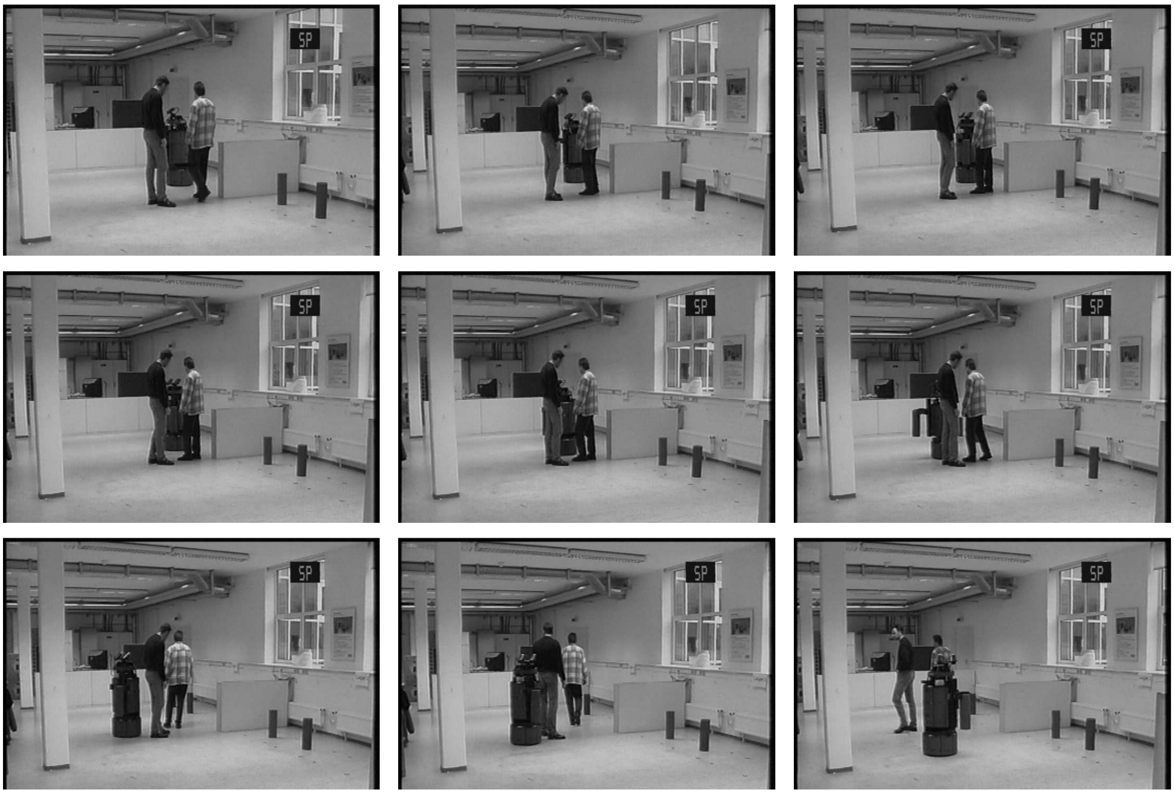


Figure 3.26: *Causing the robot with a deployed manipulator to circumvent persons.*

waypoint. Thus, the robot is able to turn into room three by passing the narrow doorway as shown in image (5/3).

3.7.5 Moving with a Deployed Manipulator in the Robotics Evaluation Area

Figure 3.32 shows the robot executing a pick-and-place task in the robotics evaluation area at the basement of *FAW*. The robot approaches the narrow opening to the hallway and due to the evaluation of the intermediate waypoints of the geometric path planning component, the robot slows down. It then moves tightly along the corner to enter the hallway with the desired heading to proceed towards the next room.

3.7.6 Grasping a Disc from a Table

Figure 3.33 shows the robot performing an approaching maneuver. The goal point has to be approached with the specified heading since turning in place would cause the manipulator to crash into the discs. The approaching maneuver uses two configured shapes to be able to approach the table. Prior to the approaching maneuver, the pose of the table and the poses of the discs are measured by means of the vision system. The overall accuracy of the vision based measuring and the approaching maneuver is such that the robot is able to grasp a disc with a diameter of 150 mm with a gripper with an opening of 200 mm.

3.8 Conclusion

Of course, the loose coupling of a simplified global path planner and a reactive component can neither ensure shortest paths nor can one ensure that all intermediate waypoints can be reached by the reactive component. There exist obstacle configurations that require complex maneuvers which can be performed only by a corresponding motion planning and execution system. Due to ignoring the kinematic constraints and due to an optimistic obstacle growing, the geometric path planning component can even return paths that are not executable by the mobile robot. However, all these circumstances are detected by the motion execution component and can thus be reported to the *sequencing layer* that then decides on how to proceed.

However, the reactive component can ensure safe operating states under any circumstances. In far most of the situations, the trajectories resulting from the loosely coupled reactive component and the simplified path planner are even close to the optimal trajectory. Such a combination is able to handle most of the standard situations in motion control and obstacle avoidance without suffering from the simplifications of the standard reactive approaches nor the computational requirements of full fledged configuration space planners. Therefore, one neither has to resign reactivity nor safety nor the ability to circumvent local minima.

An important aspect of the loose coupling of the geometric path planning component and the motion execution component is their concurrent activity so that each can contribute to the overall progress according to its responsibilities during the whole navigation task.

The configurability of the objective function allows to select among different motion control strategies and even in unknown environments one can drive safely. The chosen shortening heuristic plays a crucial role in the interaction of the geometric planning component and the motion execution component.

The overall navigation approach trades off reactivity, safety and optimality in such a way that it represents a balanced approach for most of the standard settings but at the price of not being able to reach a goal in any case. Its main strength is the ability to operate safely in a cluttered and changing environment while only imposing a very low computational load on the system and while still resulting in paths that are close to the optimal one.

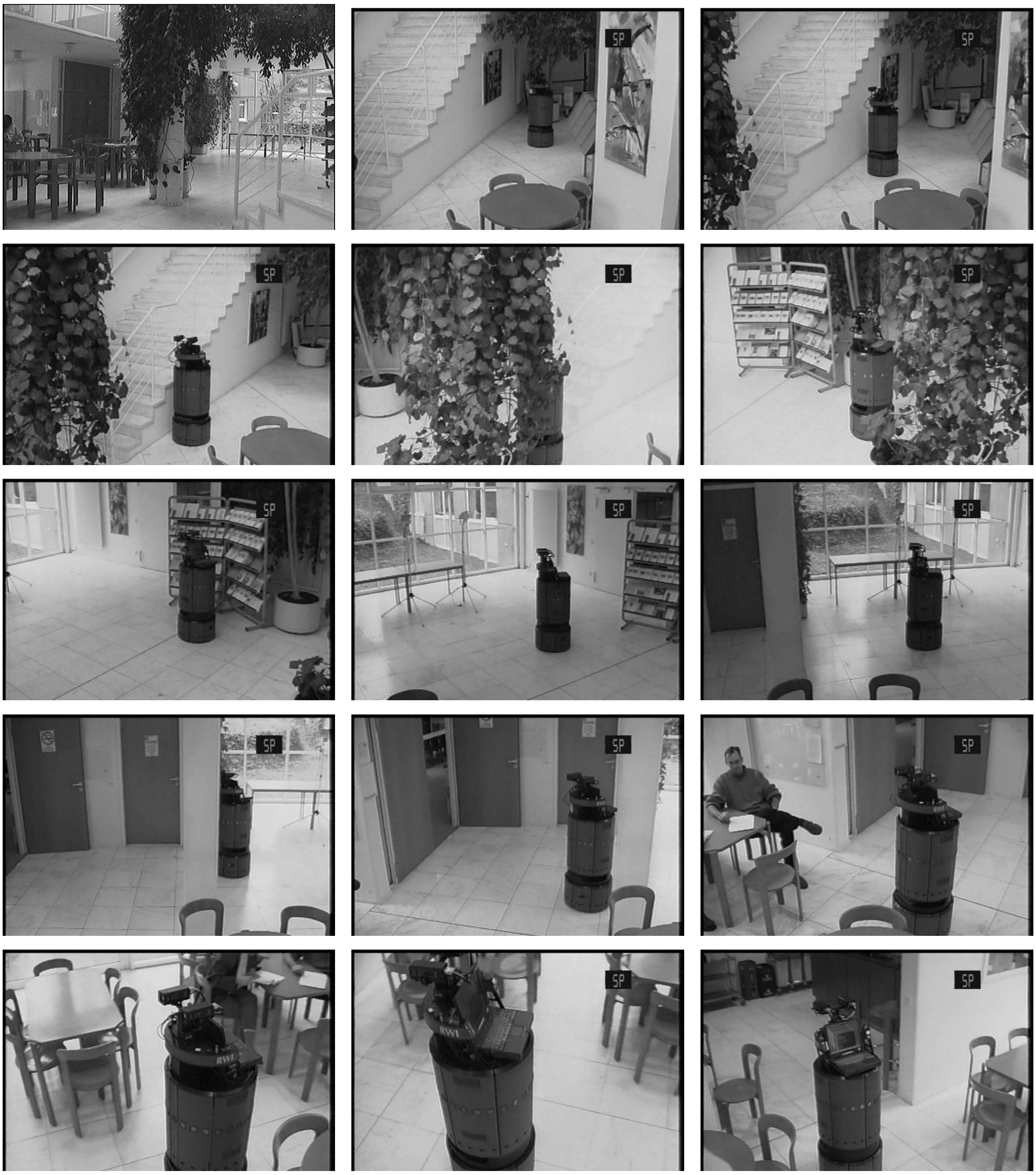


Figure 3.27: *Moving in the cafeteria of FAW.*

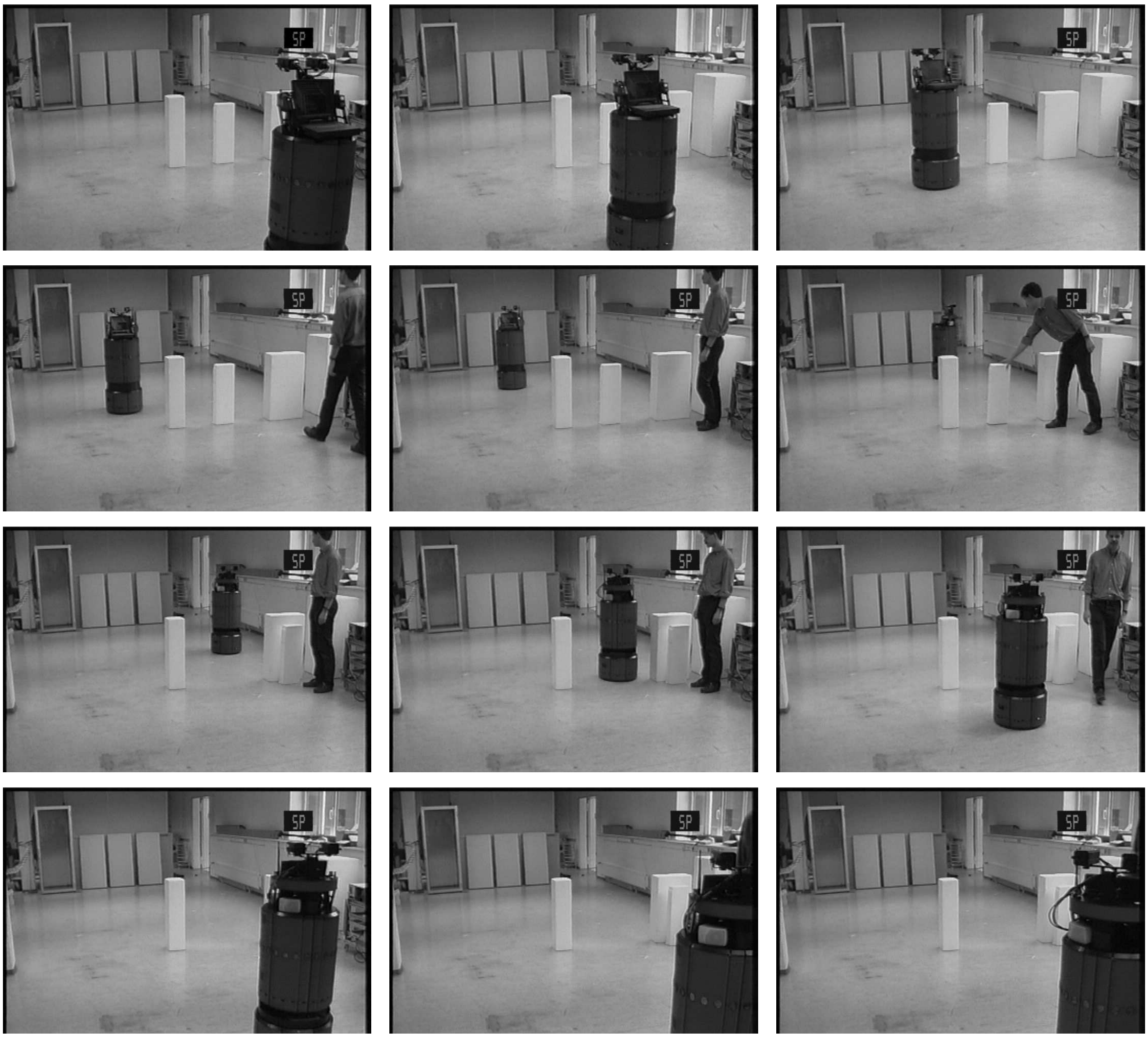


Figure 3.28: *Taking the opportunity of an opened passage.*

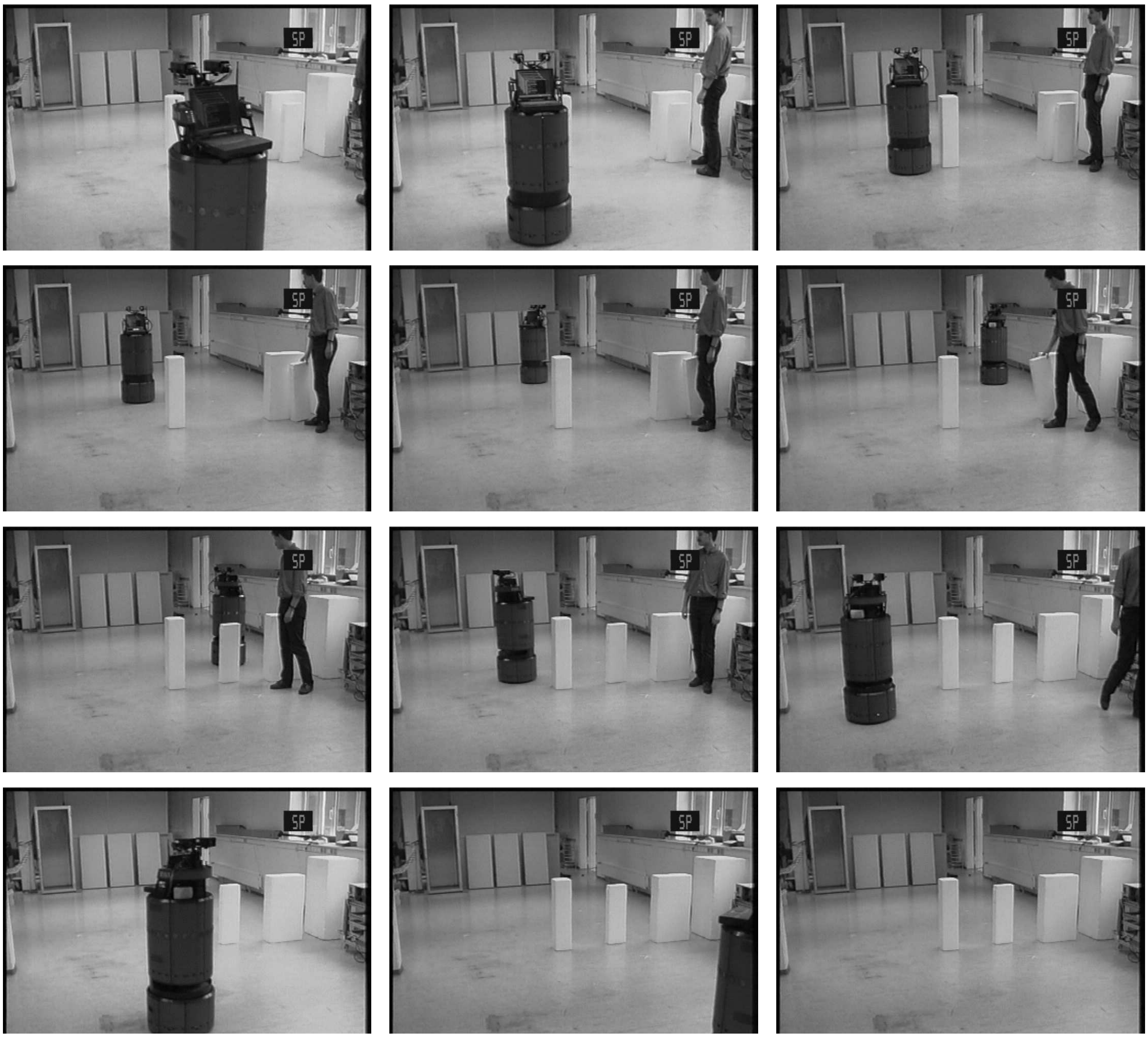


Figure 3.29: *Provoking the robot to circumvent the obstacles by blocking the passage.*

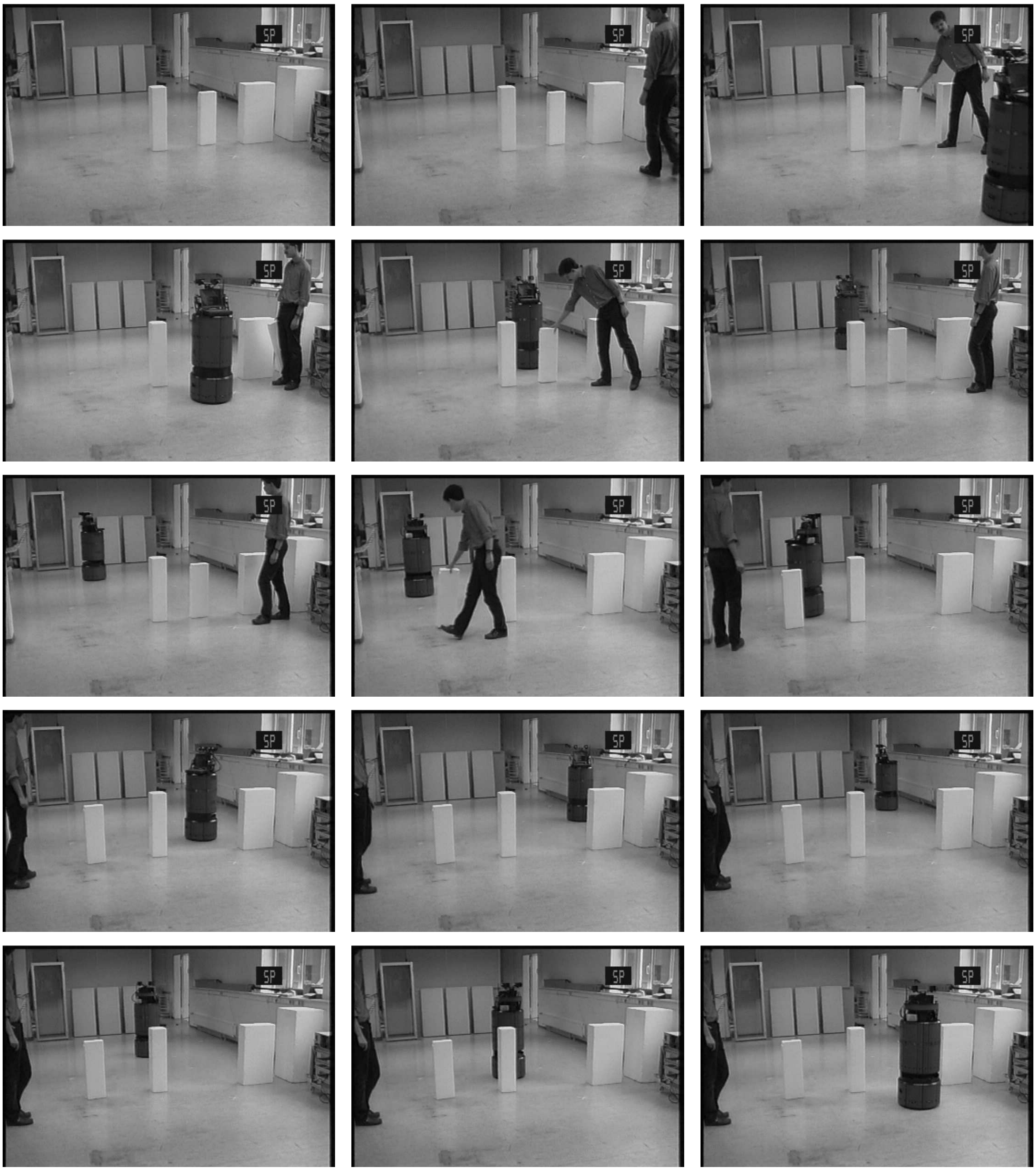


Figure 3.30: *The effect of the dynamic constraints.*

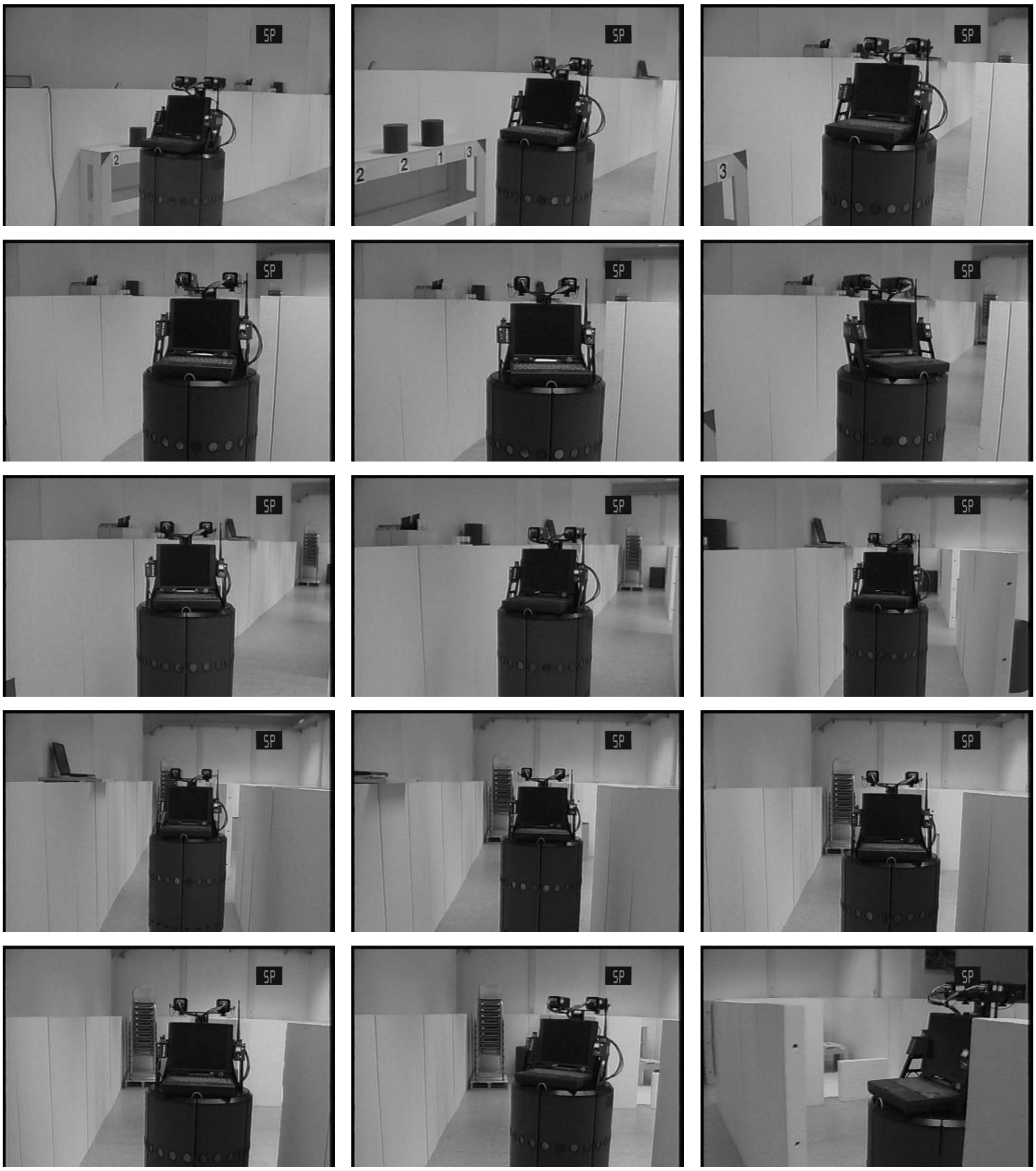


Figure 3.31: *Moving with high speed in the robotics evaluation area.*

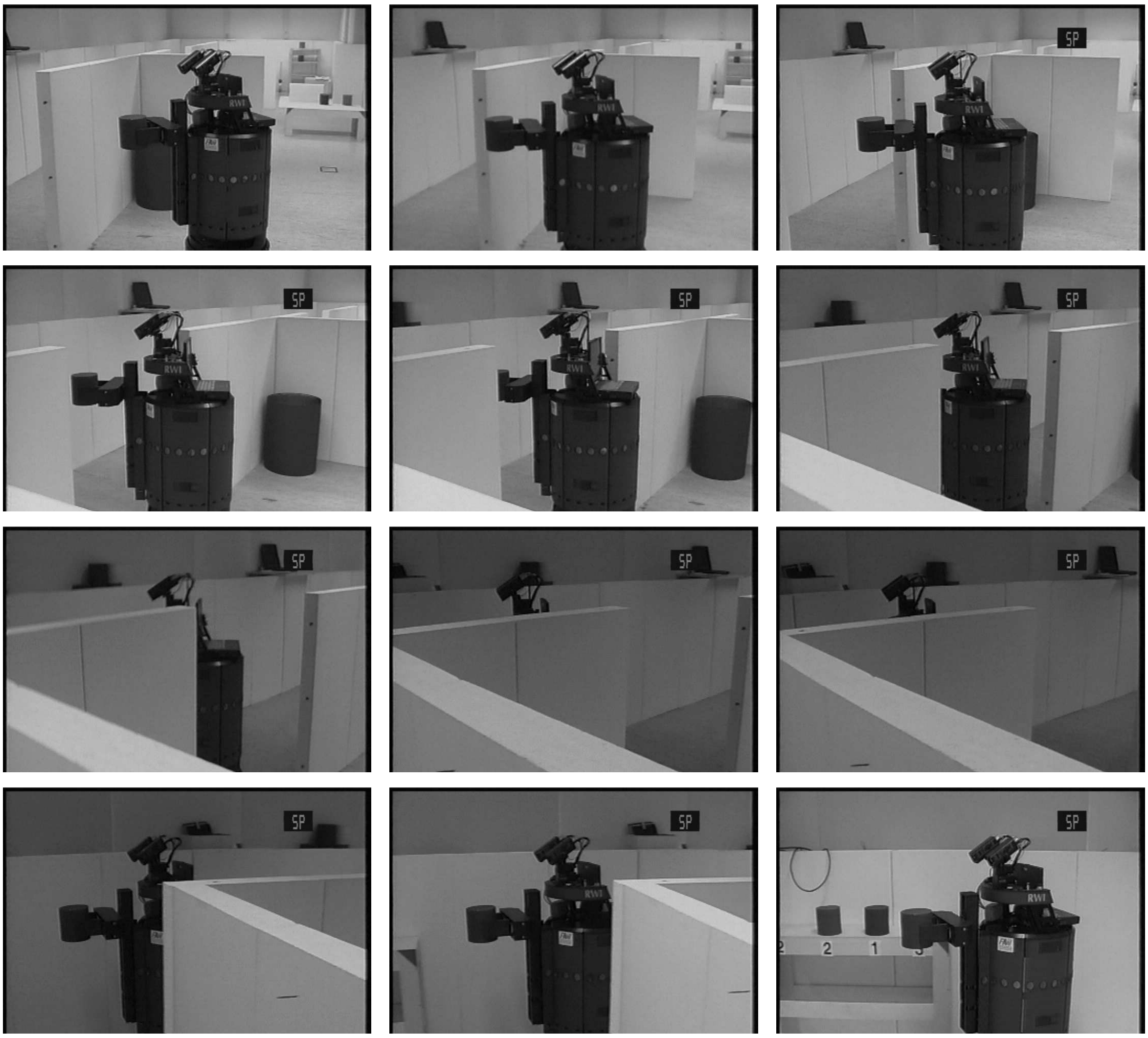


Figure 3.32: *Moving with a deployed manipulator in the robotics evaluation area.*

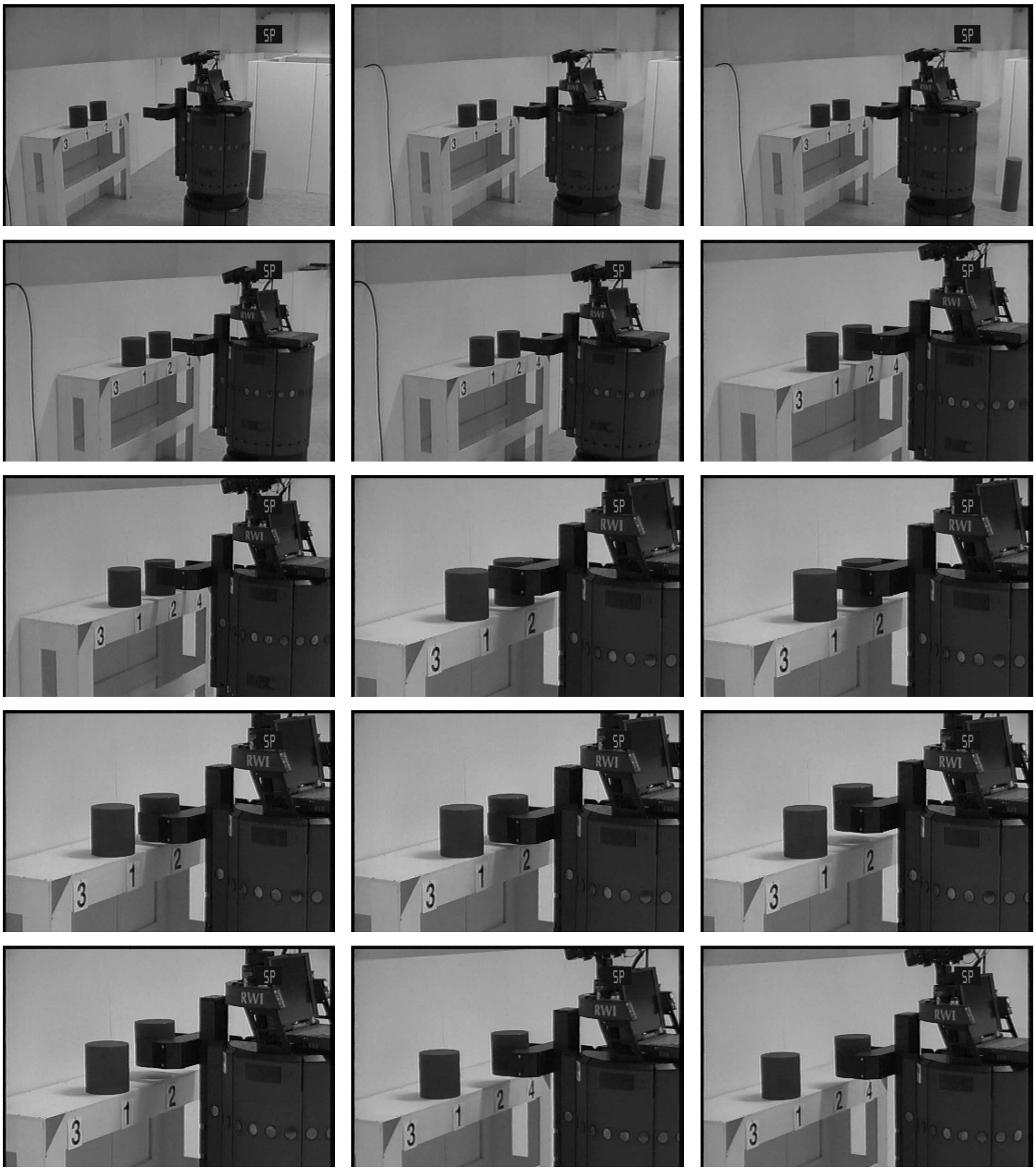


Figure 3.33: *Grasping a disc from a table.*

Chapter 4

Localization and Mapping Outline of a Computation Scheme

4.1 Introduction

Navigation tasks require knowledge about the own position and the goal position. Furthermore, the relationship between both has to be known. There are numerous ways to represent the position of a robot and the adequate approach depends on the environment and the task of the robot. A *topological* position can consist of the designator of a room, for example, and a topological map solely represents the connectivity structure of regions. For many tasks in robotics, however, a metric representation is most adequate where locations are represented by coordinates in a global frame of reference. The *pose* is the two-dimensional position in cartesian coordinates including the heading.

As soon as the robot moves greater distances, dead reckoning is not sufficient anymore and external references are necessary to keep the pose error bounded. External references can be provided in many different ways ranging from a *global positioning system (GPS)* that already provides absolute poses over artificial landmarks to appearance based methods. The latter use raw sensor data as signatures and determine the pose by aligning current sensor data with signatures.

A *global positioning system* neither provides the required accuracy nor is it available in indoor environments. Rather, sensor values provide information about the environment and are matched against a model of the environment to determine the pose of the robot. The model of the environment is called *map* and its representation depends on the kind of sensors and external references used for localization.

A map can be acquired in many different ways. In case of artificial landmarks, for example, one can use a theodolite to get the reference positions after placing the markers. Then, however, the initial effort of deploying the mobile robot in a new environment is inacceptable for many applications. The obvious idea is to use the robot itself to build the map. With appearance based methods, there is even no other option and even with many feature based approaches, it is nearly impossible to produce a map in any other way than by the robot itself. However, the map can be built accurately only if the robot knows its position.

In principle, localization and mapping is a *chicken and egg* problem. Localization requires a map and for map building the pose of the robot has to be known. There are two different approaches to tackle this problem. The first approach is to build an *absolute* map and the second one to build a *relative* map.

An *absolute* map encodes the absolute positions or poses of landmarks. These can be beacons, features or signatures. Most approaches employ a statistical view where the pose of the robot and the positions or poses of landmarks are represented by stochastic variables. The solution to the simultaneous localization and map building problem (*SLAM*) is to estimate the vehicle pose and the positions or poses of the landmarks jointly [143]. The robot pose and the positions or poses of all landmarks form the state vector and simultaneous mapping and localization is the task of maintaining an estimate of the state vector and the corresponding covariance matrix. Most implementations use the *Extended Kalman Filter (EKF)* for updating [143] [100].

However, stochastic mapping scales poorly which is known as *map scaling problem*. The storage costs are $O(n^2)$ with n the number of mapped landmarks. The computational costs of each update step are at least $O(n^2)$ and a naive implementation even requires $O(n^3)$ [59]. The complexity results from the fact that one has to maintain the covariance matrix. Reobservations of landmarks are necessary to reduce the estimation error of the landmark positions. Thereby ignoring the covariance matrix results in overly confident estimates and leads to filter divergence. The covariance matrix represents the correlations between the entries of the state vector and properly assigns the information of a reobservation to the affected poses. In the limit, the covariance matrix represents a fully correlated state vector and the lower bound on the covariance associated with any single landmark is determined only by the initial pose uncertainty of the robot when first seeing the landmark [115]. The fundamental importance of maintaining the correlations of the entries of the state vector to achieve convergence of the *SLAM* process is explicitly emphasized in [22].

It now also becomes apparent why a single *grid map* [111] without making extensions cannot be used for the *SLAM* problem. As soon as one closes a loop of assumed positions, one has to propagate the detected error back through the map. However, a *grid map* solely represents the fused result and does not provide any means to sort out the contributions of individual perceptions once these are integrated into the map.

Closing a loop is also a challenge for the *EKF*-based approaches. The initial linearization is performed when adding a landmark to the state vector and it cannot be modified anymore after integration. The linearization errors often cause the map to diverge given a sufficient number of landmark updates [83] [50].

A *relative* map consists of a network of spatial relations where nodes represent the poses or positions of landmarks and edges represent the relative relations between nodes. A node can represent any kind of feature and even signatures consisting of raw laser range scans. The relative relations can be provided by odometry, beacon detection approaches or appearance based approaches for localization, for example. A *relative map* has two significant properties. First, it requires less storage since the maximum number of relative relations is given by $n(n-1)/2$ with n the number of nodes. In particular with appearance based approaches, there are far less relations since not all signatures can be related to each other by means of the signature based localization approach. Secondly, each relative relation can be updated independently of the others. Thus, the major advantage of a relative map is that one can collect all kinds of spatial relations between landmarks without having to take into account any interdependencies to other spatial relations.

However, a *relative map* is not a suitable representation for many robotic tasks and it needs to be converted into an *absolute map*. The absolute poses of the nodes can be derived by means of a *seeding point* and traversing the *relative map*. Typically, however, a *relative map* is redundant and provides alternative paths to a node. Since an estimated *relative map* possesses errors and inaccuracies, the obtained absolute poses depend on the chosen path. In principle, it makes sense to place the nodes such that the residual error with respect to the mapped relations is minimized.

A *relative map* has significant advantages with respect to obtaining a globally consistent map. One can solve for all absolute poses at once and one can also handle non-linearities much better than with an *EKF*-based approach for an *absolute map*. Since a *relative map* maintains spatial relations between nodes, closing a loop is not a problem as long as one is able to correctly insert the closing link. Converting the *relative map* into an *absolute map* immediately results in new absolute node poses that are placed such that the error detected at the time of closing the loop is correctly assigned to node poses.

The *relative map* is the natural representation for appearance based approaches. The graph of spatial relations between reference frames links the signatures. The corresponding *absolute map* places the signatures such that these give a consistent model of the environment. One class of appearance based approaches stores laser range scans in the nodes and obtains the spatial relations by *scan matching* [104] [105] [106]. The advantage of the scan matching approach is that it does not depend on specific features in the environment.

This chapter presents an outline of a computation scheme for simultaneous localization and mapping. The foundations of the applied *covariance intersection* approach [58] as relevant for robot navigation appear to be a topic of ongoing research [81].

4.2 The Problem

Simultaneous localization and map building requires the absolute poses of the signatures. In case of incrementally building a map, new signatures and spatial relations to already mapped signatures are added to the *relative map*. Links to already mapped areas influence the absolute poses of the signatures. In particular, closing a loop requires the recalculation of the absolute node poses to compensate the pose jump at the closing link. Since mapping is done with a *relative map* and localization by means of an *absolute map*, the missing link is the conversion of the *relative map* into an *absolute map*. The problem is that calculating the absolute node poses requires to solve an error minimization problem. The complexity is such that the absolute node poses are not easily determined in parallel to map building. Thus, the robot would have to stop each time a new link is added to perform the necessary calculations before it could proceed with acquiring the next signature.

Lets consider a network of $n+1$ nodes $X_0, X_1, \dots, X_n$ where every node represents a pose vector $X_i = (x_i, y_i, \varphi_i)$. A link $D_{ij} = X_j \ominus X_i$ between the nodes X_i and X_j represents a measurable difference of the two poses. Of course, D_{ij} is nonlinear when considering poses but can be linearized in the standard way by means of Taylor expansion. We refer to D_{ij} as the *measurement equation* and furtheron only consider the linear case with $D_{ij} = X_j - X_i$. An observation of D_{ij} is modelled as $\bar{D}_{ij} = D_{ij} + \Delta D_{ij}$ where ΔD_{ij} is a random Gaussian error with zero mean and known covariance C_{ij}. According to [105], the goal is to derive the estimates of the poses given the set of measurements $\bar{D}_{ij}$ and given the covariances C_{ij} by combining all the measurements. Furthermore, the covariance matrices of the estimated poses X_i have to be derived based on the covariances C_{ij}.

As described in [104], one criterion of optimal estimation is that of *minimum variance*. The node poses X_i are determined in such a way that the conditional joint probability of the derived pose differences D_{ij} given their observations $\bar{D}_{ij}$ is maximized. In case of mutually independent and Gaussian observation errors, that is equivalent to minimizing the following Mahalanobis distance.

$$W = \sum_{i,j} (X_j - X_i - \bar{D}_{ij})^T C_{ij}^{-1} (X_j - X_i - \bar{D}_{ij}) \tag{4.1}$$

As explicated in [104, pages 86-88], the measurement equation can be expressed in matrix form as $\mathbf{D} = \mathbf{HX}$ where $\mathbf{X}$ is the $3n$-dimensional vector of the concatenation of $X_1, X_2, \dots, X_n$, $\mathbf{H}$ is the incidence matrix with all entries being 1, -1 or 0 and $\mathbf{D}$ is the concatenation of the pose differences

D_{ij}. Without loss of generality, we set $X_0 = 0$ as seeding point and $X_1, X_2, \ldots, X_n$ are relative poses from X_0. Then, W can be represented in matrix form where $\bar{\mathbf{D}}$ is the concatenation of the observations $\bar{D}_{ij}$ for the corresponding D_{ij}. $\mathbf{C}$ is the covariance of $\bar{\mathbf{D}}$ which is a square matrix consisting of the covariance matrices C_{ij} as sub-matrices.

$$W = (\bar{\mathbf{D}} - \mathbf{H}\mathbf{X})^T \mathbf{C}^{-1} (\bar{\mathbf{D}} - \mathbf{H}\mathbf{X}) \tag{4.2}$$

The solution for $\mathbf{X}$ which minimizes W and the covariance of $\mathbf{X}$ are given by

$$\begin{aligned} \mathbf{X} &= (\mathbf{H}^T \mathbf{C}^{-1} \mathbf{H})^{-1} \mathbf{H}^T \mathbf{C}^{-1} \bar{\mathbf{D}} && (4.3) \\ \mathbf{C}_{\mathbf{x}} &= (\mathbf{H}^T \mathbf{C}^{-1} \mathbf{H})^{-1} && (4.4) \end{aligned}$$

In general, one has to deal with $3n \times 3n$ sized matrices and solving the linear equation system requires $O(p^3)$ with $p = 3n$. This can be reduced to $O(p^2)$ by using a sparse matrix solver but then the covariance matrix is not available. The covariances give valuable information about the pose uncertainty during exploration. In case of linearizations, the procedure is applied iteratively using the intermediate results as new linearization points.

Due to the complexity, a different approach is required to calculate the absolute poses of the nodes. Of course, starting from scratch each time is not necessary and in the linear case, an iterative and relaxation based update rule can be derived from equation 4.1. The index i is iterating over all nodes linked with X_j.

$$\begin{aligned} W &= \sum_j {\sum_i}' (X_j - X_i - \bar{D}_{ij})^T C_{ij}^{-1} (X_j - X_i - \bar{D}_{ij}) && (4.5) \\ \nabla W_j &= 0 \quad \Longrightarrow \quad X_j^{t+1} = \left({\sum_i}' (C_{ij}^{-1}) \right)^{-1} {\sum_i}' C_{ij}^{-1} (X_i^t + \bar{D}_{ij}) && (4.6) \end{aligned}$$

Since W is a quadratic term, the iterative scheme converges to the single minimum. In the linearized case, approximately 5 % of the linearization points are updated in parallel to the relaxation process which then typically converges to a configuration close to the global optimum. However, the relaxation converges only very slowly and relaxation schemes typically require $O(n^2)$ updates to reduce the error by a constant factor [45]. Nevertheless, a relaxation based approach considerably saves computational resources by not always starting from scratch and is thus a viable approach to address the complexity issue.

The update rule shown in equation 4.6 only provides the pose estimate of a node and no covariance matrix for the estimate. The covariance matrix is not only relevant with respect to knowing about the current pose uncertainty during exploration, but also provides valuable information when closing a loop. In that case, the uncertainties of the poses of the neighboring nodes give information on where to move the node under consideration. Thus, knowing about the uncertainties of the neighboring nodes can accelerate the relaxation when closing a loop. However, the covariance matrix can not easily be determined by a local update rule since updating a node introduces cross covariances between nodes which therefore are not representing independent information anymore. Successively updating the nodes with a Kalman filter based on local information immediately results in pose estimates with zero covariance and therefore leads to overly confident estimates. Taking into account the cross covariances, however, requires global knowledge for performing a local update. Thus, the problem is to find a computation scheme that provides a conservative covariance matrix in case the cross covariances are unknown. Such a computation scheme can be applied with a local update rule for a relaxation based pose estimation.

4.3 Related Work

Of course, there is a huge body of work available in the field of filter design and sensor fusion as well as on various aspects of localization, map building and simultaneous localization and map building.

Sensor fusion in the presence of unknown cross covariances has already been addressed by [149] and [82]. Unknown or constrained cross correlations are also addressed in [66] [65]. Foundations of sensor fusion are explicated in [153] and [64].

Important results on the structure of SLAM can be found in [32], [33] and [115]. The classical approach is to use an extended state vector that contains the absolute poses of the nodes and the vehicle pose. Due to the complexity of the update step of the EKF, many efforts focus on reducing this computational complexity. Update schemes range from a pessimistic solution [80] over various kinds of decoupling [101] [78] [23] [59] to postponement of updates [92]. In [148], for example, the sparsity of the information matrices is exploited. The stability of covariance inflation methods is addressed in [81]. In general, EKF based approaches suffer from the initial linearization error when adding a feature to the state vector. Problems also arise with particle based SLAM [110] where large uncertainties require a very large number of particles.

An approach for pose tracking and map building based on dense laser range scans without extracting features is described in [104] [105] [106]. The original approach has been extended and compared to other pose tracking methods [61]. An extension to incremental mapping of cyclic environments is presented in [60]. A better way to estimate the covariance matrix with scan matching is presented in [7].

In [35], the slowly converging update rule shown in equation 4.6 is used for mapping. However, this rule does not allow for local calculations of covariances as stated in [34]. In [49], a multigrid approach for accelerating relaxation-based SLAM is proposed. It provides an interesting solution to solving the equation system in an incremental and fast way. Again, no covariance matrices can be obtained. However, using a *relative map* has the advantage that one does not suffer from initial linearization errors as is the case with EKF based approaches.

In principle, even though the general structure of the SLAM problem is well understood, it is still a field of ongoing research mainly focusing on kinds of mechanisms to address the complexity issue.

4.4 The Approach

The *relative map* is considered as *probabilistic filter network* where each node and each edge are endowed with a random variable. The filter network is interpreted in such a way that incoming estimates at a node are *fused* to result in a better estimate for the node under consideration. The chosen filter is also known as *covariance intersection* [82]. It belongs to a class of filters that perform convex combinations of random variables in the information space for fusion purposes. The most prominent example of this class of filters is the Kalman filter [86].

The following notation is used throughout this section. A n-dimensional column vector $(a_1, \ldots, a_n)$ is denoted by $\mathbf{a}$, a $n \times n$ matrix by $\mathbf{A}$. $X_a = \{\mathbf{a}, \mathbf{P_{aa}}\}$ is a random variable with expected value $E(X_a) = \mathbf{a}$ and covariance $\operatorname{cov}(X_a) = \mathbf{P_{aa}}$. In the 2-dim case the covariance of X_a and the cross covariance between the random variables X_a and X_b are denoted by

$$\begin{aligned}
\mathrm{cov}(X_a) &= \begin{pmatrix} var(a_1) & \mathrm{cov}(a_1, a_2) \\ \mathrm{cov}(a_2, a_1) & var(a_2) \end{pmatrix} && (4.7) \\
\mathrm{cov}(X_a, X_b) &= \begin{pmatrix} \mathbf{P_{aa}}\, \mathbf{P_{ab}} \\ \mathbf{P_{ba}}\, \mathbf{P_{bb}} \end{pmatrix} && (4.8)
\end{aligned}$$

Since we only consider covariance matrices, we only have to deal with symmetric positive definite matrices. We write $\mathbf{A} \succeq \mathbf{B}$ for two positive definite matrices if $\mathbf{A} - \mathbf{B}$ is positive semidefinite.

The subsequent sections are mostly taken from the publication [129]. The accordant paragraphs are not marked explicitly.

4.4.1 Filter Design and Covariance Intersection

In the following, convex combinations of random variables in the information space are considered.

Definition 1 (convex combination) *Given two random variables X_a and X_b, then their convex combination X_z in information space with $\alpha \in [0, 1]$ is calculated as follows:*

$$X_z \overset{\text{def}}{=} \Big(\alpha\, \mathrm{cov}(X_a)^{-1} \,+\, (1-\alpha)\, \mathrm{cov}(X_b)^{-1}\Big)^{-1} \cdot \Big(\alpha\, \mathrm{cov}(X_a)^{-1}\, X_a \,+\, (1-\alpha)\, \mathrm{cov}(X_b)^{-1}\, X_b\Big)$$

To clarify the presentation, the following abbreviation is introduced:

$$M \overset{def}{=} \alpha\, \mathrm{cov}(X_a)^{-1} \,+\, (1-\alpha)\, \mathrm{cov}(X_b)^{-1} \tag{4.9}$$

First, the expected value $E(X_z)$ and the covariance $\mathrm{cov}(X_z)$ of the new estimate X_z are calculated.

$$\begin{aligned}
E(X_z) &= E\Big[M^{-1}\,\Big(\alpha\, \mathrm{cov}(X_a)^{-1}\, X_a \,+\, (1-\alpha)\, \mathrm{cov}(X_b)^{-1}\, X_b\Big)\Big] \\
&= M^{-1}\,\Big(\alpha\, \mathrm{cov}(X_a)^{-1}\, E(X_a) \,+\, (1-\alpha)\, \mathrm{cov}(X_b)^{-1}\, E(X_b)\Big) && (4.10)
\end{aligned}$$

$$\begin{aligned}
\mathrm{cov}(X_z) &= \mathrm{cov}\,\Big[M^{-1}\,\Big(\alpha\ \mathrm{cov}(X_a)^{-1} X_a + (1-\alpha)\mathrm{cov}(X_b)^{-1} X_b\Big)\Big] \\
&= M^{-1}\mathrm{cov}\,\Big(\alpha\ \mathrm{cov}(X_a)^{-1} X_a + (1-\alpha)\mathrm{cov}(X_b)^{-1} X_b\Big)\, M^{-1} \\
&= M^{-1}\,\Big[\mathrm{cov}(\alpha\ \mathrm{cov}(X_a)^{-1} X_a) \\
&\qquad + \mathrm{cov}((1-\alpha)\mathrm{cov}(X_b)^{-1} X_b) \\
&\qquad + \mathrm{cov}(\alpha \mathrm{cov}(X_a)^{-1} X_a, (1-\alpha)\mathrm{cov}(X_b)^{-1} X_b) \\
&\qquad + \mathrm{cov}((1-\alpha)\mathrm{cov}(X_b)^{-1} X_b, \alpha \mathrm{cov}(X_a)^{-1} X_a)\Big]\, M^{-1} \\
&= M^{-1}\,\Big[\alpha^2 \mathrm{cov}(X_a)^{-1} + (1-\alpha)^2 \mathrm{cov}(X_b)^{-1} \\
&\qquad + \alpha(1-\alpha)\mathrm{cov}(X_a)^{-1}\mathrm{cov}(X_a, X_b)\mathrm{cov}(X_b)^{-1} \\
&\qquad + \alpha(1-\alpha)\mathrm{cov}(X_b)^{-1}\mathrm{cov}(X_b, X_a)\mathrm{cov}(X_a)^{-1}\Big]\, M^{-1} && (4.11)
\end{aligned}$$

The fusion update obviously depends on the cross covariances. Thus, the following modification of the covariance update is considered. Those fusion equations are also known as *covariance intersection* [82] or *gaussian intersection*.

Definition 2 (covariance intersection, CI) *Given two random variables X_a and X_b, then covariance intersection X_c with $\alpha \in [0,1]$ is calculated as follows:*

$$\begin{aligned} \mathrm{cov}(X_c) &\overset{def}{=} M^{-1} \\ E(X_c) &\overset{def}{=} M^{-1}\Big(\alpha\,\mathrm{cov}(X_a)^{-1}\,E(X_a)\;+\;(1-\alpha)\,\mathrm{cov}(X_b)^{-1}\,E(X_b)\Big) \end{aligned}$$

The expected value $E(X_c)$ of the covariance intersection is the same as $E(X_z)$ of the convex combination. However, the covariance $\mathrm{cov}(X_c)$ is defined independently of the cross covariances. We have to prove that $\mathrm{cov}(X_c)$ never gets overly confident on the true covariance $\mathrm{cov}(X_z)$. This is a very pessimistic approach since always highest correlations are assumed even in the case of fully independent input variables. Of course, a random variable is not just a combination of expected value and covariance. However, for filtering applications, one can substitute the currently known covariance by one which makes the estimate more worse.

Lemma 1 (covariance enclosure) *The covariance* $\mathrm{cov}(\mathrm{X_a})$ *is smaller in all directions than* $\mathrm{cov}(X_b)$ *if and only if* $\mathrm{cov}(X_b) \succeq \mathrm{cov}(X_a)$.

Proof: So-called k-sigma contours provide a convenient graphical representation of a random variable X_a. The k-sigma contour of X_a is defined by the points

$$(\mathbf{x} - E(X_a))^T \mathrm{cov}(X_a)^{-1}(\mathbf{x} - E(X_a)) = k$$

This term defines an ellipse for two dimensions respectively an hyperellipsoid for higher dimensions. The covariance $\mathrm{cov}(X_a)$ is smaller than $\mathrm{cov}(X_b)$ in all directions if the corresponding k-sigma contour of X_a is completely enclosed by the k-sigma contour of X_b. This is equivalent to

$$\begin{aligned} (\mathbf{x}-\mathbf{c})^T\mathrm{cov}(X_a)^{-1}(\mathbf{x}-\mathbf{c}) &\geq (\mathbf{x}-\mathbf{c})^T\mathrm{cov}(X_b)^{-1}(\mathbf{x}-\mathbf{c}) \\ \mathrm{cov}(X_a)^{-1} &\succeq \mathrm{cov}(X_b)^{-1} \\ \mathrm{cov}(X_b) &\succeq \mathrm{cov}(X_a) \end{aligned}$$

■

The last step holds due to the following lemma [71, page 471]:

Lemma 2 *If* $\mathbf{A}$ *and* $\mathbf{B}$ *are positive definite, then* $\mathbf{A} \succeq \mathbf{B}$ *if and only if* $\mathbf{B}^{-1} \succeq \mathbf{A}^{-1}$.

We can now show that the covariance $\mathrm{cov}(X_z)$ is always smaller in all directions than the covariance $\mathrm{cov}(X_c)$ by proving that under all circumstances $\mathrm{cov}(X_c) \succeq \mathrm{cov}(X_z)$ holds. Of course, α is the same for calculating both $\mathrm{cov}(X_c)$ and $\mathrm{cov}(X_z)$

Lemma 3 (covariance bound) *For any cross covariance with* $\alpha \in [0,1]$*:* $\mathrm{cov}(X_c) \succeq \mathrm{cov}(X_z)$.

Proof:

$$\begin{aligned}
\mathrm{cov}(X_c) - \mathrm{cov}(X_z) &= M^{-1} - M^{-1}\Big[\alpha^2 \mathrm{cov}(X_a)^{-1} + (1-\alpha)^2 \mathrm{cov}(X_b)^{-1} \\
&\qquad + \alpha(1-\alpha)\mathrm{cov}(X_a)^{-1}\mathrm{cov}(X_a, X_b)\mathrm{cov}(X_b)^{-1} \\
&\qquad + \alpha(1-\alpha)\mathrm{cov}(X_b)^{-1}\mathrm{cov}(X_b, X_a)\mathrm{cov}(X_a)^{-1}\Big] M^{-1} \\
&= M^{-1}MM^{-1} - M^{-1}[\ldots]M^{-1} \\
&= M^{-1}[M - \ldots]M^{-1} \\
&= \alpha(1-\alpha)M^{-1}\Big[\mathrm{cov}(X_a)^{-1} + \mathrm{cov}(X_b)^{-1} \\
&\qquad - \mathrm{cov}(X_a)^{-1}\mathrm{cov}(X_a, X_b)\mathrm{cov}(X_b)^{-1} \\
&\qquad - \mathrm{cov}(X_b)^{-1}\mathrm{cov}(X_b, X_a)\mathrm{cov}(X_a)^{-1}\Big] M^{-1} \\
&= \alpha(1-\alpha)M^{-1}\mathrm{cov}\left(\mathrm{cov}(X_a)^{-1}E(X_a) - \mathrm{cov}(X_b)^{-1}E(X_b)\right) M^{-1} \\
&= \alpha(1-\alpha)M^{-1}KM^{-1}
\end{aligned}$$

with $K = \mathrm{cov}(\ldots)$ positive definite. The result of $M^{-1}KM^{-1}$ is positive definite, too, since K is symmetrically multiplied on both sides. Finally $\alpha(1-\alpha) \geq 0$ holds.

■

It now is known that these fusion equations can never lead to overconfident results even when cross covariances are not known at all. However, one also has to show that the fusion equations lead to improvements of the estimate since otherwise using them for fusion makes no sense at all. A decrease of the covariance can be ensured by a proper selection of α.

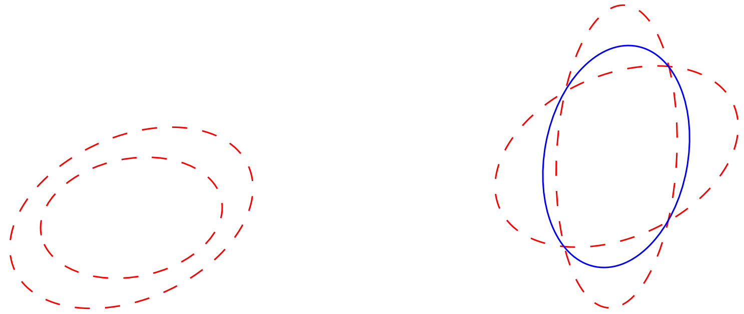


***Figure 4.1:** The resulting ellipse always encloses the intersection independently of α. It also always goes through the intersection points of the original ellipses. The* dashed *ellipses are the input ellipses and the* solid *ellipse is the result of fusion. In the left figure, the smaller ellipse is also the fusion result.*

We only consider covariances centered around the same arbitrary vector. The rationale behind a common center is that both random variables can be translated by fixed vectors without affecting the covariance matrices. Straight forward comparisons between covariance matrices are only possible as long as either $\mathrm{cov}(X_a) \succeq \mathrm{cov}(X_b)$ or $\mathrm{cov}(X_b) \succeq \mathrm{cov}(X_a)$ holds. Then the corresponding k-sigma contour of either X_a or X_b is completely contained in the other one as shown in figure 4.1. For all other cases, the ellipsoids for inverses of the covariance matrices trivially intersect as shown on the right side of figure 4.1. In this case, it is a reasonable behavior of a filtering algorithm that the covariance of the new estimate encloses the intersection as tight as possible. It can be shown that the convex combination of the inverses of the covariance matrices as used by the covariance intersection exactly shows this behavior.

To determine the best estimate, the tightest ellipsoid enclosing the intersection has to be found. The following is explicated for ellipses but the same also holds true for ellipsoids. The ellipse to the level c of the convex combination has the volume

$$\begin{aligned} &\mathrm{vol}\left(E_{\alpha\mathrm{cov}(X_a)^{-1}+(1-\alpha)\mathrm{cov}(X_b)^{-1}}(c)\right) \\ &\qquad = \pi\frac{c^2}{\sqrt{\det(\alpha\mathrm{cov}(X_a)^{-1}+(1-\alpha)\mathrm{cov}(X_b)^{-1})}} \\ &\qquad = \pi c^2\sqrt{\det\left[(\alpha\mathrm{cov}(X_a)^{-1}+(1-\alpha)\mathrm{cov}(X_b)^{-1})^{-1}\right]} \end{aligned} \tag{4.12}$$

The smallest ellipse of this type is specified by

$$min_{\alpha\in[0,1]}\det\left[(\alpha\mathrm{cov}(X_a)^{-1}+(1-\alpha)\mathrm{cov}(X_b)^{-1})^{-1}\right] \tag{4.13}$$

An iterative solution to solve for α is available in [58]. A closed form solution for the two-dimensional case has been derived by the co-author of [129].

One still has to show that the selected α always leads to a decrease of uncertainty. Since the amount of uncertainty is compared by the volume of the resulting ellipsoid, one has to show that after each update the following holds true:

$$\det(\mathrm{cov}(X_c)) \leq min(\det(\mathrm{cov}(X_a)), \det(\mathrm{cov}(X_b))) \tag{4.14}$$

Since one gets $\mathrm{cov}(X_c) = \mathrm{cov}(X_a)$ with $\alpha = 0$ and $\mathrm{cov}(X_c) = \mathrm{cov}(X_b)$ with $\alpha = 1$ and since the value for α is searched in the interval $[0,1]$, one at least gets either $\mathrm{cov}(X_a)$ or $\mathrm{cov}(X_b)$. Thus, one can make sure that the uncertainty is never increased.

4.4.2 Network of Random Variables

Lets now consider a *relative map* in form of a graph $G = (\mathbf{V}, \mathbf{E})$ with $\mathbf{V}$ denoting vertices in the graph. Edges are denoted by $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$. E_{ij} denotes the directed arc between node V_i and node V_j. The graph needs not to be fully connected and can contain loops. The set of all immediate predecessors of a vertex V_j is denoted $P(V_j)$ so that $P(V_j) := \{V_i \in \mathbf{V} | E_{ij} \in \mathbf{E}\}$. Each node V_i is endowed with a random variable X_i and each arc E_{ij} with a random variable X_{ij}. Such a network is called a *probabilistic filter network.*

The filter network is interpreted in such a way that incoming estimates at a node V_j via its immediate predecessors $P(V_j)$ are fused to result in a better estimate for node V_j. The new estimate X_j^{t+1} of node V_j is calculated as $X_j^{t+1} = CI(X_i^t \oplus X_{ij}, X_j^t)$ with $\oplus$ denoting the compounding operator. The result of $X_i \oplus X_{ij}$ is the vertex obtained when transforming X_i via the edge X_{ij} taking into account the uncertainty of X_{ij}. The transformation function to get from node V_i to V_j is thereto linearized. The node to be updated is selected randomly. The best estimate of a node is then again propagated to the connected nodes. Fusion is performed by covariance intersection since this avoids running into problems due to unavailable cross covariances of incoming estimates. Furthermore, the updates of the various nodes can be performed in any order and with different rates.

Of course, iteratively updating randomly selected nodes raises the question of convergence. Since this goes beyond the scope of this thesis, a short description is given why these updates lead to steady states. First, no update ever increases uncertainty in the network. Secondly, there exists a lower bound for the uncertainty of each node because the covariance intersection never gets overly confident. Therefore, the covariance at a certain node in the network does not decrease anymore as soon as it has reached that conservative bound given the incoming estimates. In the case of stable covariance values, the expected values are stable as well due to the then stable values of α and the whole network is in a steady state. This is also the global minimum with respect to the minimisation criterion applied at each node.

4.4.3 Example in 2D cartesian space

Lets now consider a simple example in two-dimensional cartesian space. Nodes represent a position $(x, y)^T$ and arcs the relative offset $(\Delta x, \Delta y)$ between two nodes. The transformation function to get from node V_i to node V_j is linear and can be written as

$$X_j = F(\mathbf{u}) = \mathbf{A}\mathbf{u} \tag{4.15}$$

with

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \tag{4.16}$$

and $\mathbf{u} = (X_i, X_{ij})^T$. Then, the expected value of the transformed random variable and its covariance matrix are given by

$$\begin{aligned} E(X_j) &= \mathbf{A}\, E(\mathbf{u}) && (4.17) \\ \operatorname{cov}(X_j) &= \mathbf{A} \operatorname{cov}(\mathbf{u})\ \mathbf{A}^T = \mathbf{A} \begin{pmatrix} \operatorname{cov}(X_i) & \operatorname{cov}(X_i, X_{ij}) \\ \operatorname{cov}(X_i, X_{ij}) & \operatorname{cov}(X_{ij}) \end{pmatrix} \mathbf{A}^T && (4.18) \end{aligned}$$

The cross covariance terms are zero since vertices and nodes are independent. An arbitrary node V_j of the network is updated by calculating the transformation $X_j^{t'} = F((X_i^t, X_{ij})^T)$ for any node $V_i \in P(V_j)$ and by then applying *covariance intersection* $CI(X_j^{t'}, X_j^t)$ to get the new estimate X_j^{t+1}. Of course, the nodes are highly correlated but this can be ignored in the local update due to the properties of the covariance intersection. The final state is reached if no update results in a change.

The example network is shown in figure 4.2. The network is built up incrementally in form of an octagon and a link is only introduced to the immediate predecessor. The pose uncertainty is increasing with the travelled distance. As soon as the loop is closed, the accumulated pose uncertainty gets reduced since the shorter path to the same node gives better estimates. In the example, nodes with the same distance to the origin show the same pose uncertainty after closing the loop and the furthermost node has the largest pose uncertainty. This is obvious since each additionally crossed edge adds further uncertainty. The variance of a node is equal to the variance of the initial node in case there is at least one path involving only perfectly known edges.

Since the local node update rule can cope with any kind of correlated input, even the network structure shown in figure 4.3 can be handled correctly by the iterative update scheme. This network structure is known as *Wheatstone bridge* in electrical circuits.

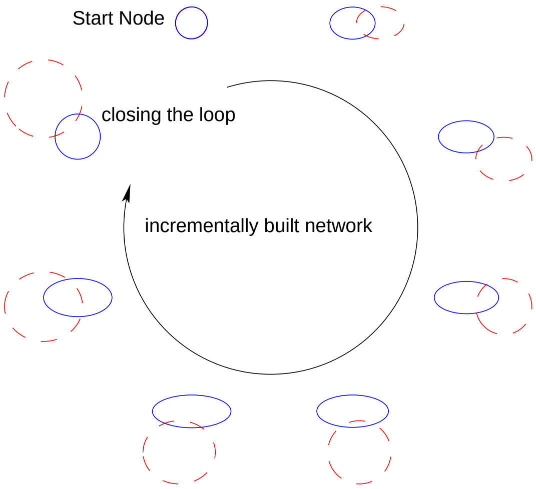


Figure 4.2: The start configuration is indicated by dashed *ellipses and the final configuration after closing the loop by* solid *ellipses. The network is built up incrementally and the pose errors are balanced as soon as the loop is closed.*

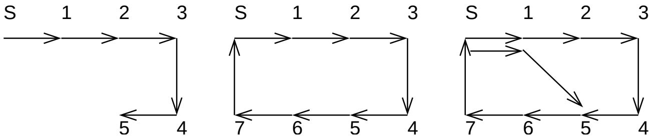


Figure 4.3: The „loop in the loop" problem. The network is built incrementally and the first loop is closed when returning to the start node S. Then the robot proceeds to node 1 and then to 5. Connecting node 1 and 5 results in two loops which have to be adjusted.

4.4.4 Application to Map Building and Localization

A probabilistic network for mapping and localization is built by means of scan matching [106]. We apply the extended scan matching described in [61]. Each vertex endows a laser scan and the pose $(x, y, \phi)^T$ of that scan. An edge represents an offset $(\Delta x, \Delta y, \Delta\phi)^T$ between two nodes.

During movement, a vehicle model is used to estimate the next pose based on odometry values. A new scan is taken and labeled by the current robot pose. The newly taken scan is added to the network by connecting it to the predecessor node. Both the relative relation based on odometry and the relative relation based on scan matching are added.

Furthermore, the newly taken scan is matched against the *anchor vertex* V_a. When starting, V_a is the first vertex in the probabilistic network. It is set to the newly added node if V_a cannot be matched anymore with the newly taken scan. The effect of this is that regions that are similar in terms of visibility all get a link to the same anchor vertex based on scan matching. It is important to note that the uncertainty of the link based on scan matching does not depend on the distance of the reference poses but only on the quality of scan matching. Thus, this approach gives much better spatial relations than the distance dependent uncertainties of the odometry. The number of remaining scan points after applying the projection filter [106] is a reliable measure to decide whether scan matching can still be applied.

A link is also introduced to an older vertex of the probabilistic network if the scan of the network closest to the current pose can be matched with the newly added scan. However, a loop is closed only if the projection filter reports a high number of points visible from both poses and if scan matching reports a low variance on the result. A much better approach that is based on comparing map patches is presented in [60].

Nodes are selected randomly and are updated applying the above described formulas. Density transformations are done by a linearized model of the link transformations using the Jacobian of the transformation equations. The new pose of the currently added node is taken as new robot pose.

Applying *covariance intersection* to a *relative map* determines node poses such that the variance of their pose is minimized. In principle, all contributions of all possible paths from the starting node to the node under consideration are fused in such a way that a conservative estimate results. The pessimistic approach is necessary since spreading the uncertainty of the initial node through a highly connected network results in highly correlated contributions to the pose of a node.

Nodes are not placed such that the residual error with respect to the measured relative distances is minimized. Minimizing the variance of the pose of a node is like placing a node according to the most certain path from the initial node towards it. However, applying *covariance intersection* provides a better result than solely considering the most certain path than it is able to exploit the remaining information even when assuming fully correlated estimates.

4.5 Conclusion

Covariance intersection allows to extract absolute node poses in the above sense from a *relative map*. During simultaneous localization and mapping, one always gets a conservative pose estimate with respect to the starting point of the exploration. It is important to note that these poses are appropriate for exploring and mapping a new environment but they do *not* represent the final *absolute map*. Nevertheless, the node poses are accurate enough for closing loops, they provide a representation of the pose uncertainty and they converge very fast towards the minimum variance pose in case of closing a loop. Since the mapping is done by means of a *relative map*, one can afterwards apply the equation solver to get the minimum mean square error solution for the absolute node poses.

Although the computation scheme is independent of how the relative map is built, the utility of the pose estimates severely depends on the covariance values. Due to the conservative approach, the covariance increases considerably fast. However, the applied scheme to extend the network based on scan matching allows to cover large distances without adding high uncertainty.

It is important to note that the *covariance intersection* can be applied only since a *relative map* is used. Applying *covariance intersection* to an *absolute map* would throw away the relations between absolute poses.

Chapter 5

The SMARTSOFT Framework

5.1 Introduction

Increasing competence of mobile robots comes along with increasing complexity of such systems. Complexity shows up at different levels of robotic systems and is mainly caused by the great number of involved components. Mastering the complexity of integration is a substantial premise towards an operational system.

Nowadays, hardware like mobile platforms, sensors and computing devices are readily available. They are also standardized at least with regard to their interfaces. Therefore, the availability of fully equipped and robust hardware platforms is not an issue anymore. This is mainly achieved by reusing approved components based on widely accepted standards as they are available in mechanical engineering, for example.

However, vital functionalities of mobile robots are provided by software and software dominance is still growing. Complex tasks can only be executed by a harmonic interaction of a diverse set of skills. Not only that single software components are of demanding complexity, but they also show elaborate interactions. Successfully mastering such complexity is an issue of the overall system architecture and is inevitably related to software engineering. Thus, both is required, a supporting framework to guide the actual implementation of robotic software components according to certain standards and an adequate concept behind that framework to ensure that the framework enforced standards are suited to master the complexity challenge.

In many respects, implementing the software of mobile robots is still a ”black art”. This is in particular worse regarding the high demands on such systems. Without an appropriate conceptual design for implementation, in the best case, one ends up with components which are not reusable on other platforms. The worst case is a complex software system which is difficult to test and maintain and is at some point even not changeable anymore without high risk.

As already successfully demonstrated in many disciplines, complexity can be mastered by component based approaches. These split a complex system into several independent units with well-formed interfaces. Complexity is reduced by restricting the focus on a single component when going into details. Fitting of components is ensured by standards for their external appearance and behavior.

The same holds true for complex software systems. In the software domain, component based approaches can reduce complexity by decoupling implementations of services from interfaces to access them. Making component dependencies explicit by well-formed interfaces supports reuse of components and simplifies assembling new systems.

It is, however, not obvious how a component based software approach for sensorimotor systems can look like and what kind of support a corresponding software framework should provide to guide and to enforce the implementation of components according to the chosen component architecture.

The major questions are what kind of standards components should obey in order to work in integrated systems, and how to design a framework such that it provides both maximum flexibility for building sensorimotor systems and guidance for a working and manageable software system. The answer is a software framework tailored to the needs of sensorimotor systems. The framework has to assist in structuring and implementing software for sensorimotor systems without restricting the robot architecture.

The presented approach for robotic software supports the implementation of interacting software components. The major contribution is a small but sufficient set of carefully chosen communication patterns for component interaction. By mapping all component interactions onto predefined patterns, all interfaces follow the predefined semantics of the communication patterns. Thus, all interfaces behave in a predefined manner. This puts the focus on services which prevent components from exposing their entire application logic and eliminates implicit component interdependencies. Another feature that goes beyond standard component approaches is the *dynamic wiring* pattern. This pattern is the basis for making both the control flow and the data flow, configurable at runtime from outside a component. It is, for example, required by the implementation of mechanisms for situated skill compositions which occur in nearly all robotic architectures.

5.2 The Problem

For a long time, the robotics community believed that integration only requires minor efforts once the needed algorithms are all available. The difficulties have been by far underestimated. Building a flexible and general robotic software system is not only a demanding but also an indispensable task towards an operational robot.

Robotics software systems are complex distributed applications. Even though computing hardware and networks get smaller and faster and can now even be integrated easily into sensorimotor systems, distributed software gets larger and more difficult to maintain and to develop. The challenges of building distributed software systems stem from both inherent and accidential complexities [15]. Inherent complexity is directly related to communication and synchronization issues. Accidential complexity stems from using unsuited tools resulting in non-extensible and non-reusable designs and implementations. One of the largest sources of complexity in general is *coupling*, resulting in non-scalability and non-distributability. Coupling means that one component is making use of knowledge about internals of other components.

Component based software development generally provides means to master the complexity issue. Often used notions of components, however, are only vague and are mainly abstracted from technical details but technical details cannot be ignored in the domain of sensorimotor systems. The requirements in terms of modularity, configurability, communication and control are critical and have to be considered altogether. Most important, existing component approaches do not provide assistance in defining the semantics of the component interfaces. However, the interface semantics is a crucial point with respect to mastering the complexity issue. This explains the need for a framework with domain specific patterns tailored to the needs of sensorimotor systems.

Another challenge of sensorimotor systems is to support the development of a system by providing a software architecture without resorting to a particular robot architecture. Components of even the same robot system often follow completely different paradigms and still have to fit together. The difficult balance of support and restriction has to be mastered in an application area where sophisticated software techniques are essential due to manifold and concurrent activities.

The goal is to build a software framework for the component based implementation of sensorimotor systems. Its purpose is to provide modular concepts and structures to cope with the complexity of robotic software systems. It has to provide patterns for often needed structures such that implemented components fit together. Critical issues are for example communication and synchronization mechanisms. The goal is neither a software engineering methodology for sensorimotor systems nor a framework which dominates every single detail at the implementational level.

5.2.1 Requirements

According to the *OROCOS* project [20] [119], several roles of users are distinguished that all put a different focus on complexity management.

End users operate applications based on the provided user interface. They focus on the functionality and use a readily provided system. They do not care on how the application has been built and mainly expect reliable operation.

Application builders assemble applications based on suitable and reusable components. They customize them by adjusting parameters and sometimes even fill in application dependent parts called *hot spots*. They expect the framework to ensure clearly structured and consistent component interfaces for easy assembling of approved off-the-shelf components.

Component builders focus on the specification and implementation of a single component. They expect the framework to provide the infrastructure which supports their implementation effort in such a way that it is compatible with other components without being restricted too much with regard to component internals. They want to focus on algorithms and component functionality without bothering with integration issues.

Framework builders design and implement the framework such that it matches the manifold requirements at its best and that the above types of users can focus on their role.

An appropriate approach has to provide consistent solutions for all of the above abstraction levels. A component based software approach per se already tackles many of the above demands. The following is a compact and widely accepted definition of a software component:

Software Component ”A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be developed independently and is subject to composition by third parties”. [112]

Note that there is an important difference between objects in object-oriented approaches and software components. The main difference is the coarser granularity of components. The definition of *objects* is purely technical and does not include notions of independence or late composition. Although these can be added to objects, components explicitly consider reusable pieces of software that have well specified public interfaces, can be used in unpredictable combinations and are stand alone entities. Figure 5.1 shows one of the advantages of using a component based approach in robotics.

The acceptance of a framework inevitably depends on how well it supports the various users and on how well it addresses the specific needs of the application domain. We therefore consider some requirements of the addressed robotics domain, which are not all per se covered by standard component based approaches.

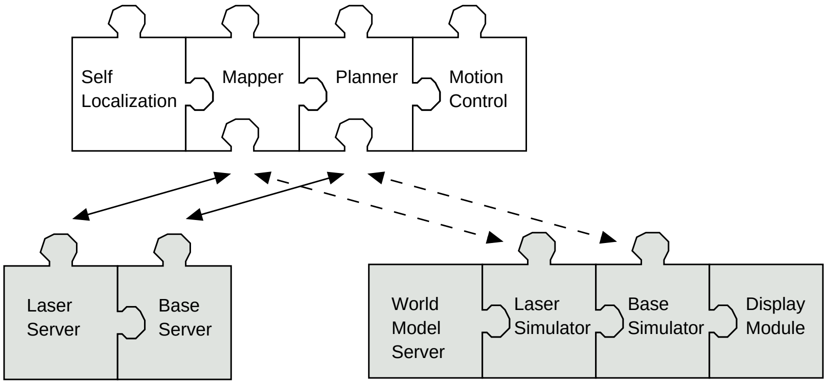


Figure 5.1: *Easy replacement of components like the servers of the real robot by a simulation system within a component based approach.*

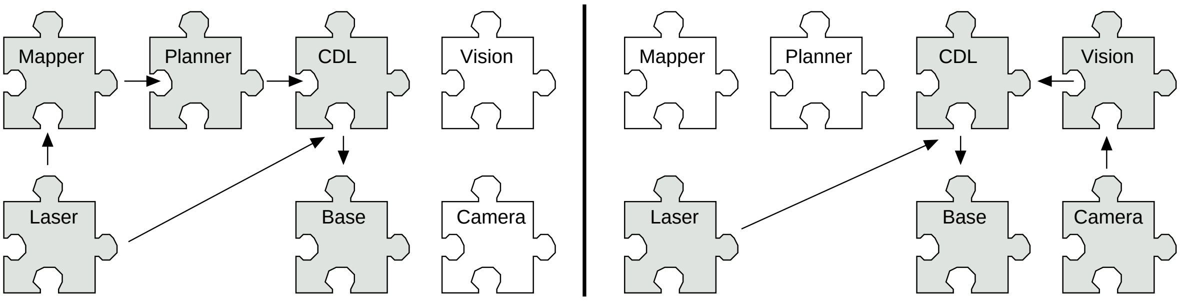


Figure 5.2: *Dynamic wiring supports task dependent composition of skills to behaviors.*

Dynamic wiring can be considered as *the* pattern of robotics. Dynamic wiring allows changes to connections between components to be made at runtime. It is the basis for making both the *control flow* and the *data flow*, configurable from outside a component which is for example needed for situated and task dependent compositions of skills to behaviors as shown in figure 5.2. Components are reused in different settings and fulfill different tasks depending on their interconnections. Since dynamic wiring is needed in nearly all robotic architectures, it has to be provided as core pattern by a robotic framework. Reconfigurable components are modular components with the highest degree of modularity. Most important, they are designed to have replacement independence. Many component approaches only provide a deployment tool to establish component connections before the application is started.

Component interfaces have to be defined at a reasonable level of granularity. The main aspect of proper component interfaces is to restrict spheres of influence such that no unwanted dependencies are propagated across component boundaries. As shown in figure 5.3, too fine-grained component interfaces can result in unmanageable software systems with closely coupled components. The figure on the left shows spaghetti-like dependencies with insight into a component whereas the figure on the right side shows puzzle-like replacement of components where internals are fully decoupled from the externally visible interfaces. A reasonable level of component interfaces, for example, avoids fine-grained intercomponent interactions and supports loosely coupled components.

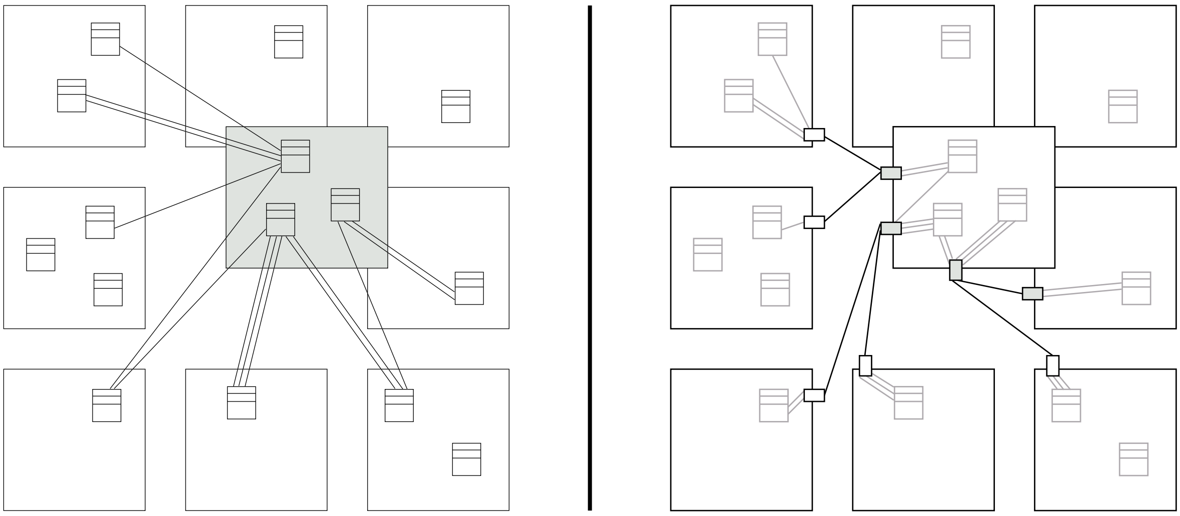


Figure 5.3: *Component interweaving with fine-grained component interfaces versus the proposed more abstract service based interfaces. Replacing a component in the service based approach requires matching of the component interfaces at the level of the proposed communication patterns instead of matching individual object member functions with arbitrary user defined semantics.*

Asynchronicity Patterns of a robotic framework should make use of asynchronicity wherever possible and should simplify usage of asynchronous interactions by the framework user as far as possible. Asynchronous interactions have to be applicable as simple as synchronous ones. Asynchronicity reduces latencies and cycle times by exploiting concurrency and it can avoid passing on tight timing dependencies between components. Steps otherwise executed sequentially can be interleaved or even executed in parallel avoiding unnecessary idle times when waiting for responses.

Component internal structures Components at different levels of the robot architecture can follow completely different designs. Sensor data processing components, for example, often loop through the same steps each time new sensor values arrive. An object recognition component may want to run time-consuming parameter adjustments in parallel to a recognition task. Component builders therefore ask for as few restrictions as possible with regard to component internal structures.

Transparency A framework has to provide a certain level of transparency by hiding details to reduce complexity. However, the level of transparency has to be adjusted to the robotics domain. Fully hiding all distribution aspects is undesired because that often not only results in a decrease in performance but also prevents predictability of the time needed for communication and of the use of system resources.

Easy usage Acceptance of a framework for sensorimotor systems is increased when it offers obvious additional value while avoiding steep learning curves. Providing access to state-of-the-art software technology without requiring every component builder to be a software engineering expert allows robotics experts to focus on algorithms and relieves them from the burden of software integration. Challenging topics which have to be addressed are, for example, location transparency of components and their services and concepts of concurrency including synchronization and thread safety. A framework provides additional value only if its usage is much simpler than meeting all the requirements without using a framework.

Although many activities in robotics are time-critical, not in all robotics applications hard real-time computing is a central concern. Very often, it is sufficient to have a timely responding system which monitors critical activities by watchdogs or uses anytime algorithms. The focus of interest here is on service robotic systems composed of many different components showing diverse and elaborate behaviors. Thus, it is much more important to have a strict component architecture which decouples component internals from interfaces and thereby, for example, minimizes unreproducible timing dependencies. Well-structured component interfaces based on standardized patterns with approved internal mechanisms increase predictability of the behavior of a component which definitely supports component and application builders. Components using a hard real-time framework to implement closed loop control can nevertheless be wrapped and made accessible within the proposed approach. The approach could also be extended and implemented in such a way that it explicitly matches different realtime levels but this goes beyond the scope of this thesis.

At the organizational level both component and application builders, expect means to consistently specify component services and interfaces. In particular in robotics, many equivalent solutions for subsuming functionalities to components exist and therefore clear agreements on component services and on their use are essential. Means to reach early agreement on component responsibilities and interface semantics guided by use cases simplify the detection of missing services, uncover unassigned responsibilities and support finetuning of component interactions before implementation. Afterwards, interface descriptions simplify elimination of malfunctions in the overall software system by first performing compliance checks at the component level before going into details. This is particularly useful in robotics where testing of the full functionality of a system is often very time consuming and difficult due to lack of reproducible experiments and settings.

Sensorimotor systems require components to be distributable over several networked computers. The first reason is the overall computing power needed to process all the concurrent activities. The second reason is the required low latency when one has to react to new sensor input. Therefore, components have to be transparent with respect to access and location. This covers hiding differences in data representation and hiding where a resource is located even across platforms. It is, however, important to always be able to clearly identify which service comes across which intermediary network components to properly rate delays. Therefore, migration transparency with respect to accessing other components or with respect to providing services is desired at the level of component builders. An application builder, however, normally has to finetune the distribution of components across the available resources and therefore relocation transparency is not wanted in robotics applications since this severely affects predictability.

A component often comprises several complex services to avoid extensive data exchange between components. Concurrent activities within a component with separate control flows are therefore a natural structure in robotics. In order to simplify the implementation of multithreaded components, a framework should provide approved patterns to cope with common pitfalls. These include concurrency transparency, which eliminates the need for further synchronization within components for concurrent access to shared resources. Furthermore, different components are often linked to form a control loop and need to exchange high volumes of data. Efficient component interactions are therefore crucial and a centralized communication approach like a blackboard could very soon become a bottleneck as well as fine grained communication where the bottleneck results from frequent access to attributes of remote objects.

However, one also has to recognize that a component based approach for sensorimotor systems and a supporting framework need not provide all of the functionalities of a general purpose framework. The domain dependent focus does not necessarily limit its applicability to robotics but of course weights requirements in a specific manner. In robotics, for example, one normally does not have a repertoire of components providing similar or equivalent services. In the majority of cases, mechanisms for automatic load balancing of services are as little required as world wide scalability. On the other hand, it is, for example, important to be able to control the number of concurrently spawn and active threads to properly master the bounded resources on board a sensorimotor system.

5.3 Related Work

Of course, there are examples of succesfully deployed complex robotic systems like museum tour guides [147]. These systems show impressive robustness and contain state-of-the-art algorithms. However, their overall software has often evolved over years and, unfortunately, provides no means to make implicitly coded experiences available as guideline for new system designs.

Frameworks for building sensorimotor systems are so far mainly provided by the robot manufacturers and are in most cases vendor specific. Those frameworks are primarily developed for simple access to hardware components and do not take into account requirements resulting from more complex robotic architectures. *Mobility* [74], for example, is an up-to-date and *CORBA* [26] based package provided with the platforms of ISI [75]. A robot system is represented by a set of hierarchically organized objects. Although that framework already offers much more than those provided by many other vendors, it still only provides an object centered view and therefore is mainly useful for simple access to the robot's hardware and for assembling of few skills.

MIRO [152] is another *CORBA* based middleware designed to ease the implementation of autonomous mobile robots. It is vendor-independent and its core adheres to modern design principles and software standards. Again, it is a purely object oriented approach without providing guidance with respect to specifying interfaces.

Other systems like *ARIA* [2] are object oriented application programming interfaces. These do not provide any suitable support for structuring more complex systems consisting of many different components. *Saphira* [95] runs on top of *ARIA* and is an architecture for mobile robot control. Every activity must partition its work into small and incremental routines that fit into the cycle time of the system. Finite state machines with a fixed cycle time which are triggered synchronously then control the execution order. This approach is suited only for some control algorithms and does not at all scale to more complex systems. Furthermore, *Saphira* implements a specific robot architecture with the so-called *local perceptual space* at its center. That is designed to accommodate various levels of interpretation of sensor information [96]. Therefore, it is the implementation of a specific architecture and not a more generally usable framework.

Many other robotic software systems are also implementations of specific architectures and force users to stick to the chosen approach. Others are libraries to provide specific functionalities and do not care about integration. *GenoM* [46, 108], for example, is a complete framework addressing various levels of robot architectures but expects users to adapt to the strict and predefined internal structure of components. *GenoM* focuses on the component internal architecture which is independent of the used communication mechanism. However, its requirements on a component communication mechanism are fully matched by the approach presented in this chapter. *GenoM* is based on *codels* which are uninterruptable code elements. Again, every activity is expected to be split into smaller code segments according to states of a finite state automaton. Therefore, long running calculations have to be distributed over several states to be interruptable. Of course, this makes tool support and coherence easier but also limits the component builder in choosing its favored architecture inside a component.

An aspect often neglected by available or proposed frameworks is that a sensorimotor system is composed of many and heterogeneous algorithms. Decisional processes are completely different from control loops. Components have to be coordinated and configured to generate the desired behavior and therefore have to be controllable. General purpose component architectures can of course significantly simplify the implementation of a framework for sensorimotor systems and can provide the underlying infrastructure.

Despite the wide use of component based approaches in software engineering, there is still no generally agreed notion of *component* available. Various aspects of component based architectures are considered in [122]. Others define software components as ”binary units of independent production, acquisition and deployment that interact to form a functioning system” [146]. Various definitions can also be found in [19]. Definitions of components are often too abstract and mainly address organizational issues. Abstract concepts of a component are therefore implemented in many different ways resulting in different and rivaling component architectures.

The component model of *Java* [76] enables developers to write reusable components once and run them anywhere. One can benefit from the platform-independent power of *Java*. *Java Beans* [77] act as a bridge between component models and provide seamless means for developers to build components that even run in *ActiveX* [1] container applications. *Jini* [79] uses events as wiring points which can even be used across virtual machines. These approaches are, however, all related to the *Java* technology which is not the first choice for all components in the robotic domain.

The *Simple Object Access Protocol (SOAP)* [144] is an open standard to communicate over different component models and is based on *XML* [154] and *HTTP* [72]. It has been mainly developed to ensure interoperability at the expenses of performance and is therefore not suited as general approach across all levels of sensorimotor systems.

However, neither the component model of *Java* nor the one of *SOAP* assists the component builder in defining the appropriate interface level of a component. They provide means to access and run components, but leave it to the component builder to define suitable interface objects with appropriate methods. Nevertheless, exposing arbitrary objects with arbitrary methods makes it difficult to the application builder to compose complex applications. For example, component interfaces can differ with respect to the concurrency level they support and it is up to the application builder to figure out how to safely access and use another component.

CORBA [26] is a vendor-independent standard for distributed objects. It provides mechanisms by which objects transparently make requests and receive responses as defined by the *object request broker*. It is an application framework that provides interoperability between objects built in possibly different languages and running on possibly different machines in heterogeneous distributed environments. In general, object centered middleware systems alone are not sufficient. These primarily address transparency aspects and often do not have a component concept. They nearly never provide means to ensure standard interfaces with predefined semantics.

The *CORBA* specification is being extended continuously and the number of supported features is growing rapidly. The major *CORBA* releases are illustrated in figure 5.4. Initially, *CORBA* solely supported synchronous function invocations with the main focus on object location transparency. Even interoperability between different object request broker implementations was not given. Access to remote objects on a per-attribute basis resulted in fine-grained communication and led to huge network traffic which is a severe threat to performance and scalability. This is not only a problem in robotics, but has also attained attraction in the domain of enterprise applications [68]. *Object-by-*

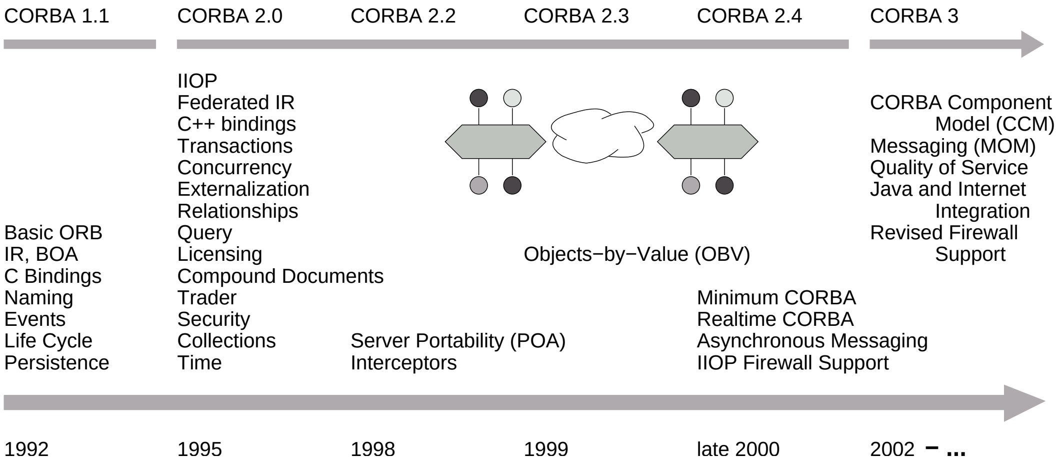


Figure 5.4: *Features of major* CORBA *releases.*

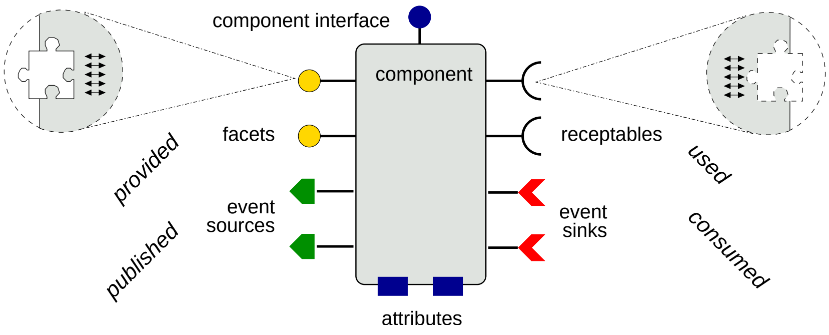


Figure 5.5: *The* CCM *provides standard component interfaces for late composition. Components are however still interweaved at the fine-grained level of user defined member functions with arbitrary semantics.*

value semantics has not been introduced until *CORBA 2.3* and *asynchronous messaging* has finally been incorporated into *CORBA 2.4* [27]. Even today, many *CORBA* implementations still have problems with the tricky details of the *valuetype* semantics or do not cover the complete asynchronous messaging interface. A good overview can be found in [123]. The latest substantial extensions led to the umbrella term *CORBA 3* with the specifications released in september 2002 [29]. Now, *CORBA* also comprises an advanced and elaborate component model called *CCM* [30] illustrated in figure 5.5. A key role is played by the *CIDL* (Component Implementation Definition Language) to describe component interfaces which are processed by a compiler. Object containers provide mechanisms for late composition. The component connections shown in figure 5.6 are established with package deployment.

The *CCM* is, however, still centered around the concept of remote objects. Late composition of components is achieved by an additional abstraction layer which allows to configure the connection between the local object proxy and its remote implementation at component deployment. The main disadvantage of an object centered view is that arbitrary member functions of user defined objects are exposed as component interface which interweaves components at the fine-grained level of arbitrary user member functions at interface objects. Full implementations of *CORBA 3* are expected within

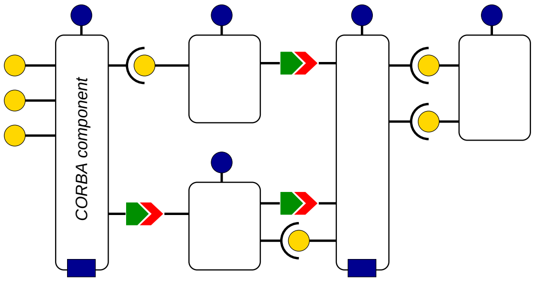


***Figure 5.6:** Assembling of components with the* CCM *at application deployment based on offered and required interfaces.*

the year 2003.

CORBA is a general-purpose and powerful approach. In the beginning, lack of features made *CORBA* unsuitable for robotics applications. The problem now is that the countless options and services show a remarkable complexity. This requires a skilled and experienced software developer to properly select and apply that part of *CORBA* which best fits the requirements of the application domain. A good example of critical issues related to the robotics domain are the object activation models. A separate thread per request can easily result in unpredictable resource consumption. *CORBA* is therefore suited as tool for framework builders but its application requires far too much software engineering expertise from other user groups of a robotic framework. Interestingly, features important for robotics applications like asynchronicity, which are standard in message oriented middleware, made their way into *CORBA* very late. The same holds true for the *object by value* semantics, which allows coarse grained communication producing traffic only once when the whole object is transmitted instead of producing fine grained interactions each time an attribute is accessed. This is in particular important in robotics, where objects can hold large data structures which are processed in complex algorithms requiring frequent object access. Furthermore, an *object by value* semantics significantly simplifies object responsibilities and is therefore the better approach in many distributed applications compared to using references.

As already mentioned, *CORBA* implementations differ with respect to both the subset of features they are compliant with and their major focus. *TAO* [133] is an advanced open source *CORBA* implementation which is both standard conformant and widely supported. Its goal is to provide an efficient high performance and scalable real-time implementation with predictable timing characteristics. Its implementation is based on the *Adaptive Communication Environment (ACE)* [132]. *ACE* is a framework for programming networked applications. It provides patterns to master the inherent complexity of concurrent networked applications [136, 137]. This very powerful framework provides operating system abstractions and is therefore a very good starting point for implementing an operating system independent robotics framework. Its patterns are, however, intended for framework builders.

Of course, middleware systems like *CORBA* provide a suitable base for implementing a software framework for robotics but they cannot guarantee a consistent robotic software system solely based on the fact that *CORBA* and its component model are being used. This would completely ignore the additionally required domain specific knowledge. The experience with various kinds of component structures, interface characteristics and suitable interaction patterns is a key part, too. One of the most important aspects of a framework and of patterns is to provide just that meta knowledge in a reusable form. Even with *CORBA*, *dynamic wiring* is not provided as self-contained service or readily applicable pattern.

The *OROCOS* project [20] was started to provide a software framework to be able to share, distribute and reuse robotic software components among laboratories or companies. The open source project wants to tackle the lack of a generic programming framework in the context of robotic systems. The approach presented in this chapter was first presented in [131] and is now incorporated into the *OROCOS* project [127].

5.4 The Approach

Complexity reduction of component based approaches results first and foremost from splitting complex systems into smaller and independent entities with well-formed interfaces. This allows the different user groups to focus on one aspect or detail at a time and to rely on the framework that afterwards still everything fits together. Complexity reduction of component based approaches is based on the general and powerful concept of *decoupling*. The different requirements on a software component approach for robotic systems are all in some aspect related to *decoupling* and therefore the proposed approach is explicitly oriented at the design goal of *decoupling* to master the complexity issue.

From the view of the component builder, decoupling for example comprises the separation of structure and functionality to ensure that a service can be implemented without caring about which and how other components will use it. Decoupling means fully hiding the component internal structure from other components. Decoupling at the level of components does not require to consider internals of other components while focusing on the details of one component. Inside a component, decoupling means that no extra care has to be taken when services are called concurrently from different threads. Decoupling of activities comprises exploiting asynchronicity as much as possible and hidden from the framework user.

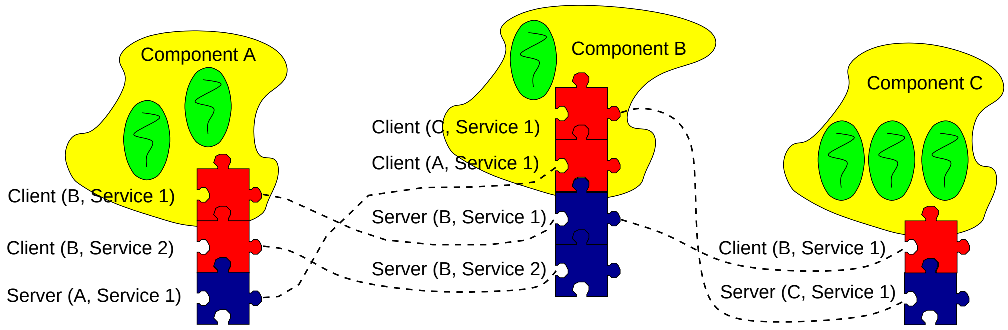


Figure 5.7: *Overview on the approach. The component interfaces are composed of only a few different standard patterns.*

Implementing a component approach based on the concept of *decoupling* requires to identify a suitable starting-point. Implementing the various aspects of decoupling requires to master the component interactions. Thus, the developed approach selects the intercomponent communication as a suitable starting-point for an appropriate implementation of a component based software framework for sensorimotor systems. The basic idea is to provide communication patterns as the only component interfaces to control all component interactions as shown in figure 5.7. *The unique feature of the approach is to compose any component interface out of a set of predefined communication patterns.* If the communication patterns are chosen carefully, one can cover all communicational needs with a very small set of patterns. These can be specified in such a way that they provide a clear semantics and that they fulfill all aspects of decoupling relevant in the robotics domain. Mapping all component interactions onto predefined and strictly specified patterns is the key approach to match the various aspects of decoupling in a component based approach for sensorimotor systems.

5.4.1 The Underlying Concept

Communication patterns squeeze every component interaction into predefined patterns. A communication pattern defines the communication mode and provides a fixed set of access functions. Two examples of a communication mode are a *one-way* interaction and a *push*-service. Access functions of a communication pattern implement approved patterns like a deferred reception of answers or an asynchronous processing of requests. Communication patterns implicitly code the knowledge about out of what kind of structures one should compose component interfaces and they provide that knowledge in an easy to use way.

Components are technically implemented as processes. A component can contain several threads and interacts with other components via predefined communication patterns. A component can provide and use any number of services. Services of components can be wired dynamically at runtime.

Communication Patterns assist the component builder and the application builder in building and using distributed components in such a way that the semantics of the interface is predefined by the patterns, irrespective of where they are applied. A communication pattern defines the *communication mode* like *one-way*, *request/response* or *push* and provides predefined *access modes* like *synchronous* and *asynchronous* service invocations or *handler based* request handlings. This allows issues of concurrency and asynchronicity to be handled inside the patterns and fully hidden from the user. Communication patterns always consist of two complementary parts, a *service requestor* and a *service provider*, representing a *client/server*, a *master/slave* or a *publisher/subscriber* relationship.

Communication Objects parameterize and bind the communication pattern templates. They represent the content to be transmitted via a communication pattern. They are always transmitted *by value* to avoid fine grained intercomponent interactions when accessing an attribute. Furthermore, object responsibilities are much simpler with locally maintained objects than with remote objects. Communication objects are ordinary objects decorated with additional member functions for use by the framework.

Service Each instantiation of a communication pattern provides a service. A service comprises the communication mode as defined by the communication pattern and the transmittable content as defined by the communication objects.

The service based view comes along with a specific granularity of a component based approach. Services are not as fine grained as arbitrary component interfaces since they are self-contained and meaningful entities and not only dependencies spanning across components. Major characteristics of the proposed service based component approach are as follows (also illustrated in figure 5.8):

- A fixed set of communication patterns which provides predefined component interaction modes is used to compose any component interface. Communication patterns provide the only link of a component to the external world and can therefore ensure decoupling at various levels. Communication patterns decouple structures used inside a component from the external behavior of a component. Decoupling starts with the specific level of granularity of component interfaces

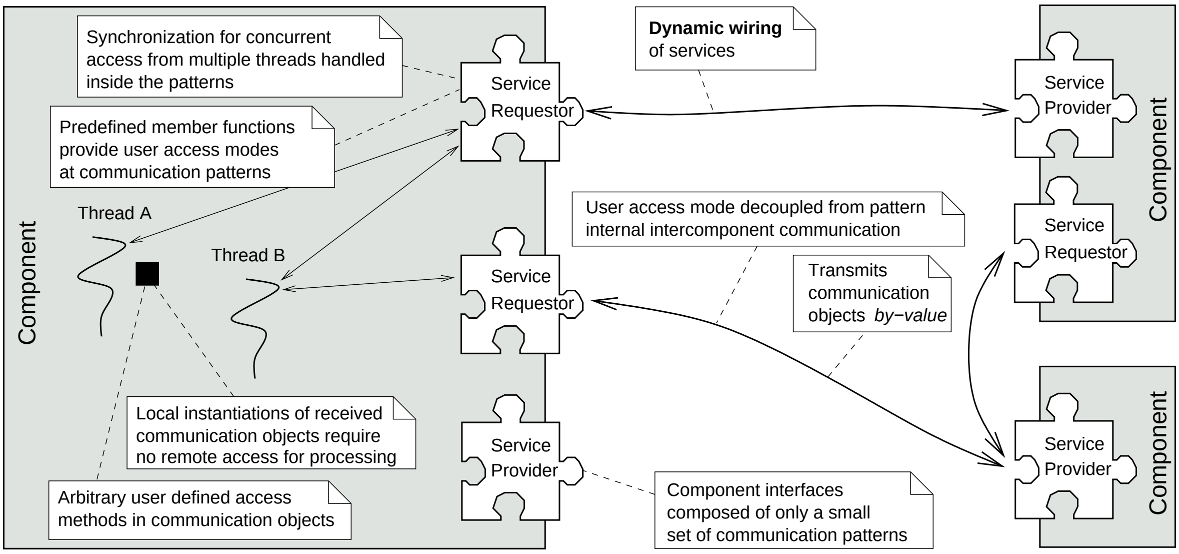

Synchronization for concurrent access from multiple threads handled inside the patterns
Predefined member functions provide user access modes at communication patterns
Thread A
Thread B
Component
Service Requestor
Service Requestor
Service Provider
Local instantiations of received communication objects require no remote access for processing
Arbitrary user defined access methods in communication objects
Dynamic wiring of services
User access mode decoupled from pattern internal intercomponent communication
Transmits communication objects by−value
Component interfaces composed of only a small set of communication patterns
Service Provider
Service Requestor
Component
Service Provider
Component

Figure 5.8: *Summary of the major characteristics. All components only interact via services based on predefined communication patterns. These not only decouple components but also handle concurrent access inside a component. Each component can provide and use any number of services.*

enforced by the communication patterns which avoids too fine-grained interactions and ends with the message oriented mechanisms used inside the patterns between components.

- Using communication patterns with given access modes prevents the user from puzzling over the semantics and behavior of both component interfaces and usage of services. One can neither expose arbitrary member functions as component interface nor can one dilute the precise interface semantics and the interface behavior. Given member functions provide predefined user access modes and hide concurrency and synchronization issues from the user and can exploit asynchronicity without teasing the user with such details.

- Arbitrary communication objects provide diversity and ensure genericity with a very small set of communication patterns. Individual member functions are moved from the externally visible interface to communication objects.

- *Dynamic wiring* of intercomponent connections at runtime supports context and task dependent assembly of components.

Since component interactions are mapped onto predefined communication patterns, all component interfaces are composed of the same set of patterns. Therefore, looking at the external interface of a component immediately opens up the provided and required services, and looking at the communication pattern underlying a service immediately opens up the usage and semantics of this service.

Figure 5.9 illustrates the standard remote object approach. The application builder can expose arbitrary member functions resulting in bulky interfaces with a user-defined semantics. The local stub and the remote object are tightly coupled. In *CORBA*, for example, it is within the responsibility of each client whether to access a server synchronously or asynchronously. The server needs not to be changed but it is then left to the component builder to properly synchronize incoming responses at the client side with the activity which initiated the request.

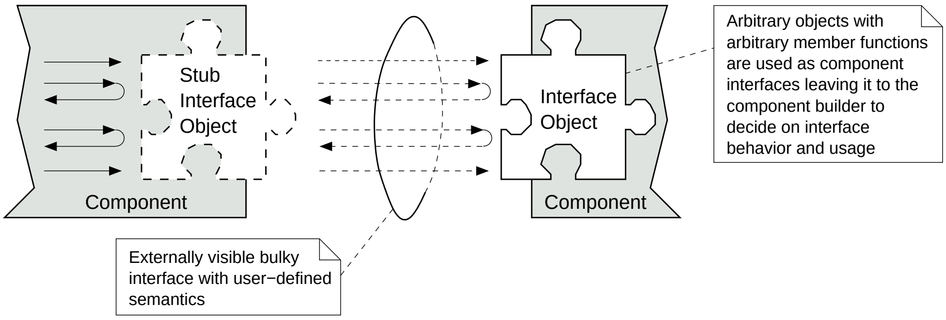

Stub
Interface
Object
Component
Interface
Object
Component
Arbitrary objects with arbitrary member functions are used as component interfaces leaving it to the component builder to decide on interface behavior and usage
Externally visible bulky interface with user−defined semantics

***Figure 5.9:** Distributed object models and various component models use arbitrary objects as interfaces.*

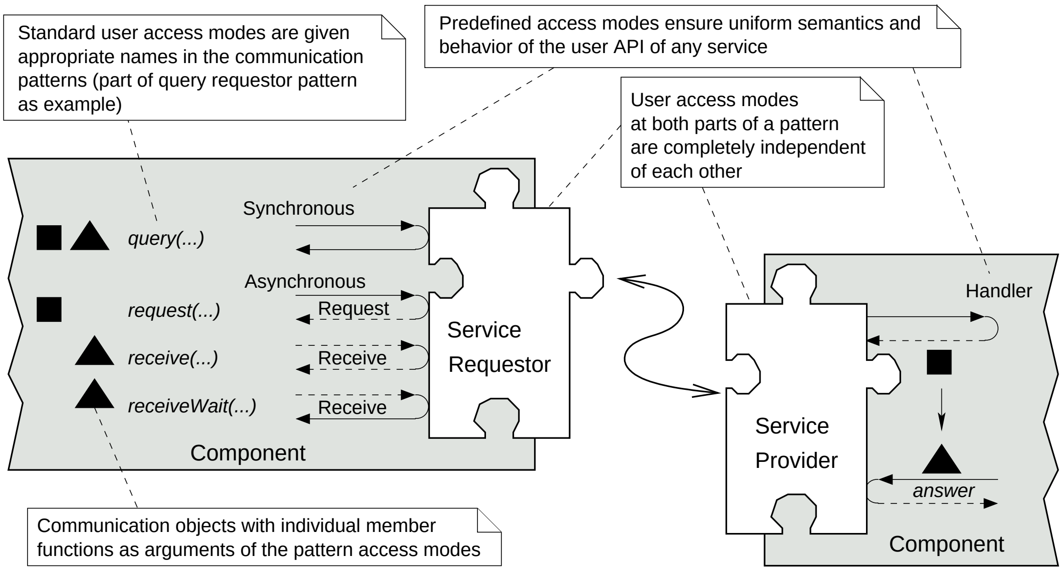

Standard user access modes are given appropriate names in the communication patterns (part of query requestor pattern as example)
Predefined access modes ensure uniform semantics and behavior of the user API of any service
User access modes at both parts of a pattern are completely independent of each other
Synchronous
query(...)
Asynchronous
request(...)
Request
receive(...)
Receive
receiveWait(...)
Receive
Service
Requestor
Component
Handler
Service
Provider
answer
Component
Communication objects with individual member functions as arguments of the pattern access modes

***Figure 5.10:** The proposed service based approach exposes services as component interface. Services are based on only a few different patterns which already comprise access modes.*

In contrast, the proposed approach moves individual member functions from the externally visible component interface to the communication objects. As illustrated in figure 5.10, services are only accessed via predefined member functions with a precisely defined semantics. The major advantage is that now asynchronicity issues and access modes are no longer part of every single object but belong to the communication patterns. Synchronization and concurrency issues can now be handled inside the patterns based on approved procedures instead of dealing with them again and again in every single remote object.

Providing access modes and not leaving it to the user to decide on the member functions to be exposed as interfaces is a significant part of the approach. This avoids puzzling over the semantics of otherwise arbitrary user defined member functions of remote objects and relieves the component builder from implementing error prone individual solutions for concurrent pattern and component access. This results in reduced component complexity and increased stability. Communication patterns significantly simplify the interface semantics problem.

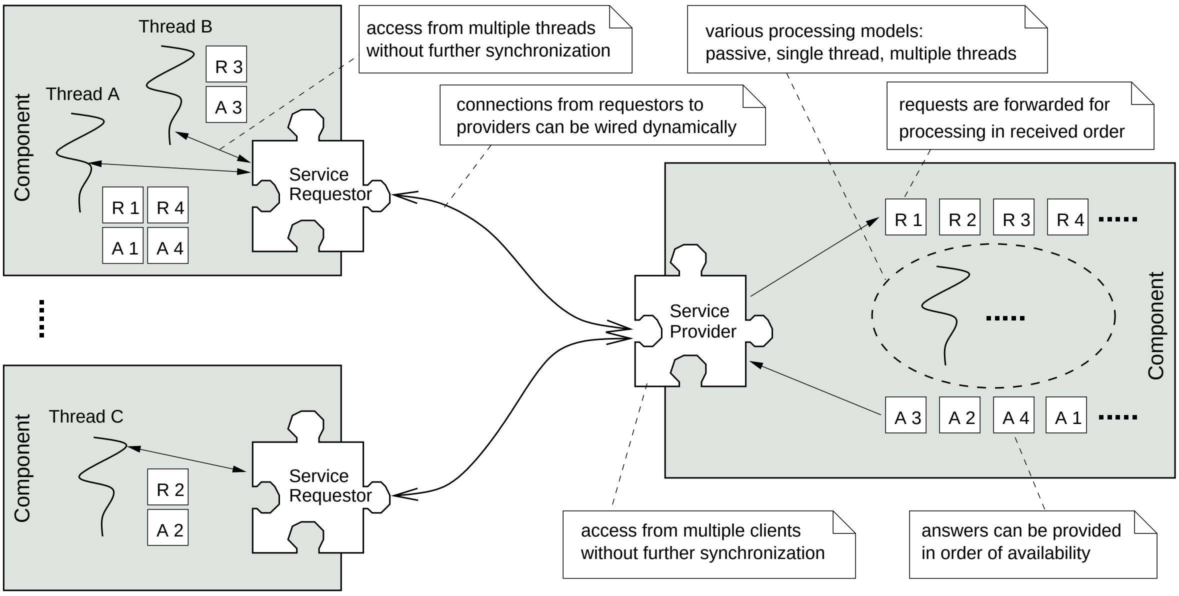

Component
Thread B
Thread A
R 3
A 3
R 1
R 4
A 1
A 4
Service Requestor
access from multiple threads without further synchronization
connections from requestors to providers can be wired dynamically
various processing models: passive, single thread, multiple threads
requests are forwarded for processing in received order
Component
Thread C
R 2
A 2
Service Requestor
Service Provider
R 1
R 2
R 3
R 4
A 3
A 2
A 4
A 1
Component
access from multiple clients without further synchronization
answers can be provided in order of availability

Figure 5.11: *Decoupling of components. The example shows a query service.*

Specifying every communication pattern as two complementary parts in combination with predefined access functions decouples both the communicating components and the access modes at both communication endpoints. As shown in figure 5.11, the interaction between the service provider and the service requestor is completely hidden from the user and decoupled from the user access modes. Since no intercomponent interaction can circumvent the communication patterns, the framework can ensure the required level of decoupling.

Both parts of a communication pattern are completely independent and decoupled stand alone entities. One part is not just a proxy to hand over member function calls to the other side. Member functions at both parts of a communication pattern are not required to be identical. Therefore, access modes can be completely different at both parts of a communication pattern and are assigned to communication patterns solely based on use cases. Asynchronous interactions are also handled inside the patterns. This includes the assignment of deferred answers to the corresponding requests in a way that is transparent to the user. Appropriate member functions of access modes simplify the usage of asynchronous interactions significantly by providing comfortable means to test or to wait for deferred answers. Decoupling access modes of the service requestor from the ones used by the service provider does not impose constraints across component boundaries. Components can implement individual processing models without being restricted to the ones used by their counterpart. This is among others an important building block for reducing complexity since components can use their individual component architectures without being affected by other components.

Dynamic wiring also requires both parts of the communication patterns to be stand alone entities with appropriate state machines. For example, the service provider has to be able to discard answers for meanwhile disconnected service requestors. The service requestor has to correctly cancel pending requests when getting disconnected from the service provider while keeping already received answers for deferred pickup.

Genericity of the approach is achieved by using arbitrary and individual communication objects. They are always transferred *by value* and are therefore always maintained locally. Accessing locally available communication objects avoids fine-grained intercomponent interactions and results in increased efficiency. The implementations of the member functions of the communication objects are neither affected by aspects of the intercomponent communication nor can these implementations introduce any intercomponent dependencies. In accordance with the service based view, communication objects can be accessed and manipulated without entailing external component interactions. They can also be used independently of the subsequent state of the service providing component once they have been obtained. Furthermore, object responsibilities and synchronization issues are much simpler with locally maintained objects than with remote objects. Arbitrary member functions defined by the user can be implemented without taking into account any aspects of intercomponent dependencies since their execution context never spans across components.

The proposed approach interweaves components at the level of services as against to interweaving components at the fine-grained level of member functions. Since all services are based on only a very small set of patterns, one can strictly control intercomponent dependencies and can therefore ensure proper component interfaces.

5.4.2 The Communication Modes

Restricting all component interactions to given communication patterns requires a set of patterns that is sufficient to cover all communicational needs. One of course also wants to find the smallest such set for maximum clarity of the component interfaces and to avoid unnecessary implementational efforts for the communication patterns. On the other hand, one has to find a reasonable trade-off between minimality and usability. The goal is to keep the number of communication patterns as small as possible without ignoring easy usage.

In principle, every kind of component communication can be realized once one-way links between components are established. This, however, is not very convenient since more complex communication modes then have to be composed manually. The component builder would then be responsible for handling concurrent requests and for coordinating incoming and outgoing messages. The basic purpose of the communication patterns is to relieve the component builder from these error-prone details by providing approved and reusable solutions. Therefore, communication patterns have to provide patterns for higher level component interactions. The communication patterns itself are of course completely independent of the internally used mechanism to transmit data between the communication endpoints of a pattern. They can be implemented on top of standard middleware systems as well as on top of simple message based systems.

Pattern	Relationship	Initiative	Service Provider	Communication Mode
send	client/server	client	server	one-way communication
query	client/server	client	server	two-way request/response communication
push newest	publisher/subscriber	server	server	1-to-n distribution
push timed	publisher/subscriber	server	server	1-to-n distribution
event	client/server	server	server	asynchronous notification
wiring	master/slave	master	slave	dynamic component wiring

Table 5.1: *The set of communication patterns.*

Analyzing the various kinds of interactions between components allows to identify a small number of interaction patterns that are listed as communication patterns in table 5.1. Since communication patterns wrap services, the two parts of a pattern are called *service provider* and *service requestor*. Even though the push patterns form a publisher/subscriber relationship, we call the publisher the server and the subscriber the client. In case of a client/server relationship and for the push service, the service provider is the server part of the pattern. In a master/slave relationship, the service is provided by the slave since the slave executes the commands of the master. In all other cases, the service provider is the server. Generally, a service provider can handle an arbitrary number of service requestors at the same time.

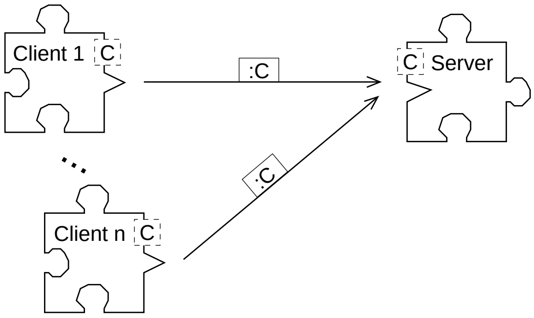


Figure 5.12: *The* send *pattern (C type of sent object).*

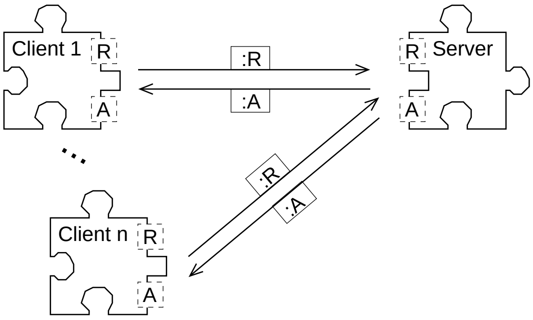


Figure 5.13: *The* query *pattern (R request type, A answer type).*

A client initiated communication can be either a one-way or a two-way communication and is covered by the *send* and the *query* pattern. The *send* pattern is shown in figure 5.12 and transmits only one communication object from a client to the server. It is for example used to control the velocity or the steering angle of a mobile platform. It can also be used to implement a data driven processing chain of components. Figure 5.13 illustrates the *query* pattern. A client sends a request containing individual parameters and receives an individual result from the server. The *query* pattern is also used if a service is needed at a very low rate compared to the cycle time of the service. To save communication bandwidth, it makes more sense to perform a query if new data is needed instead of being overrun by not needed updates. The *query* pattern is for example used to request a particular part of a map where the request specifies the size and the origin of the map patch returned by the answer communication object. The *send* and the *query* pattern cover all client initiated component interactions.

The *push* pattern provides a publish/subscribe mechanism for data distribution. As shown in figure 5.14, every client gets the same data simultaneously as soon as new data is available at the server without polling. Compared to the previous patterns, the communication is initiated by the server and not by the client. A push service is superior to polling in case several clients need the same data or in case the update rate is dictated by the server. The *push* pattern is for example used to distribute laser range scans to various components and to provide an updated local map patch as soon as something has changed. According to use cases in robotics, the *push* pattern is split into a *push newest* and a *push timed* pattern. The *push newest* server has to be fed with new data by the user and then distributes the data to all subscribed clients. The *push timed* server is triggered periodically by the framework to acquire new data and distributes the data on a regular basis with individual client update intervals. It relieves the component builder from handling timing schemes.

The *event* pattern supports asynchronous notifications. An event activation provides a parameter set for the event condition. The event condition is checked at the server side. An event activation fires as soon as the event condition becomes true under its individual activation parameters. The

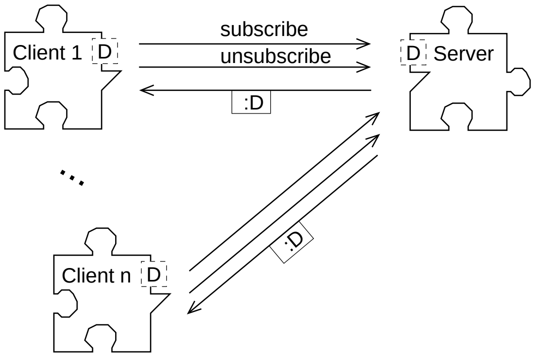


***Figure 5.14:** The* push *pattern (D distributed data).*

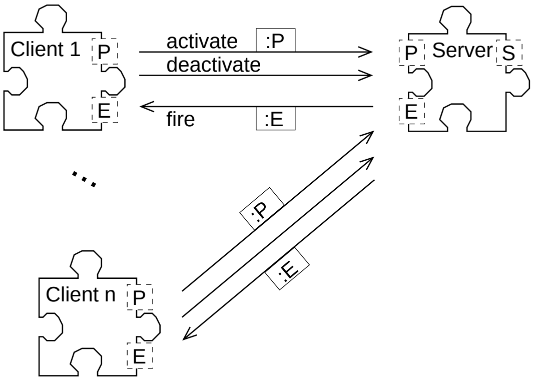


***Figure 5.15:** The* event *pattern (P activation parameter, E returned when fired, S state description checked in event condition).*

returned communication object can contain any required information about what caused the event to fire. As with the *push* pattern, the server is the active part and the server decides on when to check the event condition. The *event* pattern, however, does not distribute the same data to every subscribed client, but provides individual answers for each firing activation. Events are mainly used for vertical communication between components where they synchronize task execution progress with a discrete description of steps in a monitoring component. Events are also used to, for example, monitor the battery voltage and report when the voltage drops below critical thresholds.

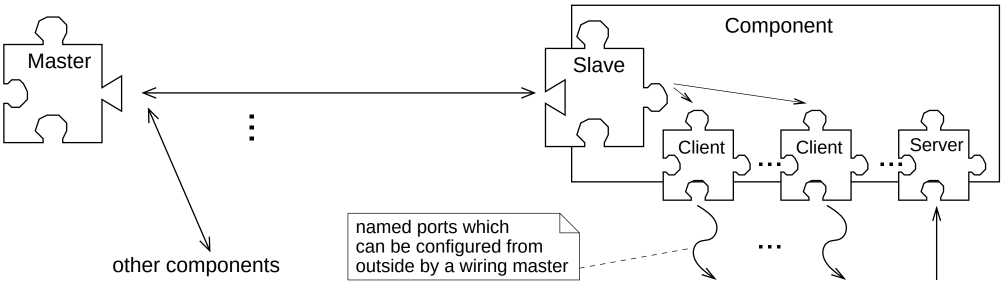


***Figure 5.16:** The* wiring *pattern. One master can control any number of slaves in parallel.*

The *wiring* pattern shown in figure 5.16 provides a consistent mechanism for dynamic wiring of client objects from outside a component. The service requesting part of a communication pattern can expose itself as a port by registering itself at the component's wiring slave object. The port can then be connected to an appropriate service provider from outside the component via the master part of the *wiring* pattern. A service requestor can connect to any service provider as long as both are compatible in terms of the service. Compatibility of a service requires that both parts belong to the same type of communication pattern and that both are parameterized by the same types of communication objects. Wiring at runtime covers every kind of changes in connections between client and server parts of services at any time and requires the communication patterns to appropriately cope with connection changes even when there are pending communication activities. Dynamic wiring is, for example, needed to change the data flow between components to compose different behaviors out of a set of skills.

The proposed set of patterns is sufficient to cover all communicational needs since even with a one-way communication as provided by the *send* pattern, one can emulate any other communication mode. With respect to usability, one should, however, provide at least further common communication patterns. The proposed set of patterns covers two-way communication by the *query* pattern, $1 : n$ distribution by the *push* patterns and conditional signaling by the *event* pattern. An additional pattern that is in particular tailored to the needs of robotic systems is the *wiring* pattern. Considering both variants of the *push* pattern, this results in only six different communication patterns for composing any component interface.

5.4.3 The Communication Objects

Communication objects parameterize the communication patterns and are transmitted *by value*. This appears as moving a copy of an object between components. The framework transmits objects by only transmitting the relevant content of a communication object. At the recipient, that content is used to reconstruct a local instance of the appropriate communication object type. The whole procedure is transparent to the user and works like a copy constructor or assignment operator across component boundaries.

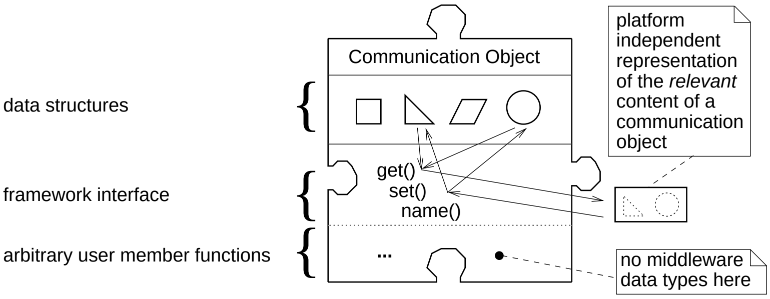

Communication Object
data structures
framework interface
arbitrary user member functions
get()
set()
name()
...
platform independent representation of the *relevant* content of a communication object
no middleware data types here

Figure 5.17: *The basic structure of a communication object with the framework interface.*

Communication objects are regular objects which are decorated by additional member functions for use by the framework. Figure 5.17 shows the basic structure of a communication object. A *get*-method extracts the relevant data structures and converts them into a platform independent representation for transmission. A *set*-method converts the platform independent representation back into the original object internal representation. Finally, every communication object type can be identified by a unique name provided by the *name*-method.

Figure 5.18 illustrates how a communication object migrates between components. The example communication pattern is parameterized by one communication object of type *C*. The user task is operating on instance *A* of the communication object of type *C*. Calling a member function of the user interface of the *service requestor instance R* passes the *communication object instance A* to the communication pattern ①. Now, the communication pattern *R* internally extracts the platform independent representation of the content of *A* and passes the platform independent representation to the communication mechanism ②. The communication mechanism of the service providing component receives the platform independent representation and forwards it to the appropriate service providing communication pattern which in our case is the instance *P* ③. The service provider fills in a local instance *B* of the appropriate communication object type *C* which can then be further processed ④. The instance *B* is completely local and accessing it does not initiate any further communication. Of

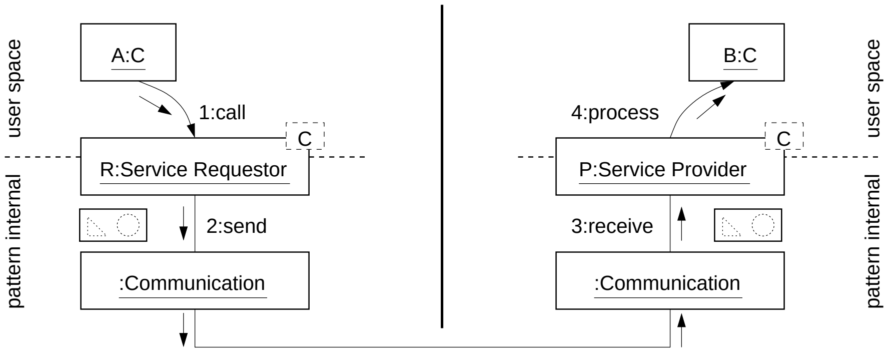


Figure 5.18: *Transmitting a communication object* by value.

course, setting internal data structures to the received content also requires the *set* member function to properly initialize the not transmitted parameters.

This approach for transmitting objects *by value* is completely independent of the communication mechanism used and can be implemented on top of any middleware system. Its major advantage is that the user accessible member functions can use arbitrary types for arguments and return values. Not being restricted to the types of the underlying communication mechanism at the user interface avoids pushing the types of the communication framework into user space as it is the case with plain *CORBA*, for example. Furthermore, this independence allows to use types of widespread libraries like the *STL* [113] in member functions of communication objects. Therefore, component builders do not need additional glue logic to mediate between the component framework and their libraries. New member functions can easily be added locally to communication objects without affecting other components or inflating the interface that is visible across components. Communication objects focus on the content to be transmitted. They remove the object member functions from the component interface and make them to a component internal issue.

5.4.4 The User Access Modes

Figure 5.19 shows several access modes. The first approach is based on *member functions* which are called and executed by the user thread. The synchronous access mode blocks until the overall communication activity is completed or aborted. The asynchronous access mode supports deferred reception of answers. The asynchronous access mode always provides a blocking and a non-blocking member function to collect deferred answers. The blocking member function suspends the calling task till the answer is available. The non-blocking member function does not suspend the calling task but requires polling to detect whether the result has yet been received and can already be fetched. Polling is executed locally based on the pattern internal state automatons and does not require any external communication.

The second approach is based on *handlers*. It is best suited to process asynchronously received requests and to implement data driven approaches. The handler is invoked by the thread of the framework and in case of a *passive* handler, it is also completely executed by the thread of the framework. *Active* handlers, in contrast, process requests in a separate thread and conserve the framework resources. As shown in figure 5.19, results of a handler are provided to the communication pattern by a separate *answer* member function and are not returned as return values by the handler. This is to decouple the handler invocation from providing return values. This significantly simplifies the imple-

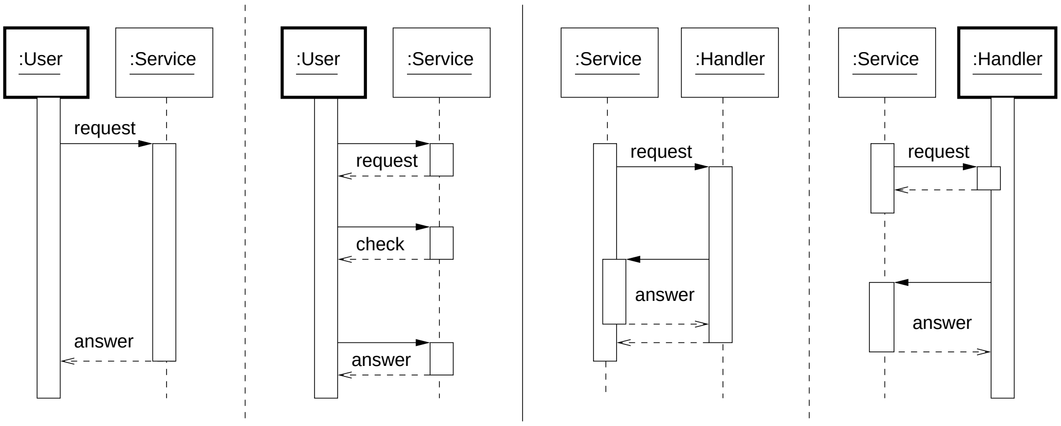


Figure 5.19: *Different access modes from left to right: User invoked synchronous and asynchronous pattern access and pattern invoked passive and active handlers.*

mentation of diverse processing models like thread factories or processing chains where the request is pipelined through various active processing units.

Pattern	Service Requestor		Service Provider		Remark
	Handler	Member function	Handler	Member function	
send	-	×	×	-	-
query	-	×	×	-	-
push newest	-	×	-	×	-
push timed	-	×	-	×	1
event	×	×	-	×	2
wiring	-	×	-	-	3

[1] Handler at service provider to trigger data acquisition and distribution.

[2] Handler at service provider to check event condition.

[3] Service provider does not require user accessible interface.

Table 5.2: *Overview on the user access modes of the communication patterns.*

The assignment of user access modes to the communication patterns is summarized in table 5.2 and illustrated in figure 5.20. The assignment is based on an analysis of use cases and drops access modes for which only very few use cases exist. Since both parts of a communication pattern are stand alone entities, one can in principle provide any possible combination of user access modes with a communication pattern. The service requestor, however, normally only requires a synchronous and an asynchronous access mode to invoke services. The *event* client additionally provides a handler based interface to support event driven architectures based on firing activations. A handler would also make sense for the *push* client but it turned out that a data driven architecture with the push service is implemented easier using a separate thread blocking on the member function based interface.

Service providers normally implement a handler based interface for easy handling of incoming service requests. In case of the *push newest* and the *event* service, the activity of the service provider is, however, not determined by asynchronously received requests. In fact, the *push newest* service is activated each time new data is provided that is to be distributed. The *event* service provider is

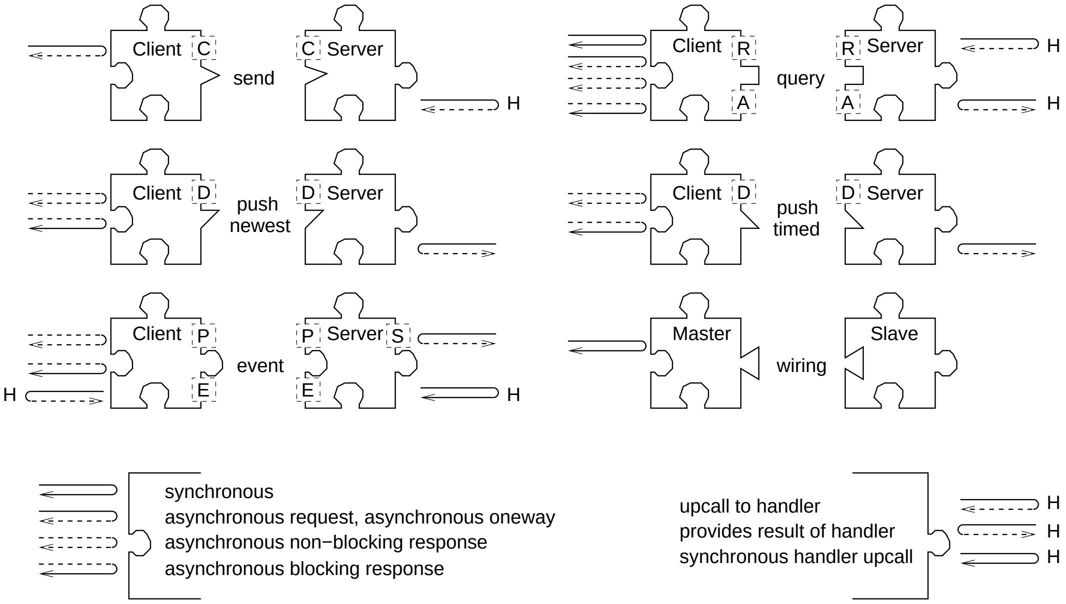


Figure 5.20: *Visualization of the user access modes assigned to the communication patterns.*

activated with every newly provided state description. A new state description initiates the testing of the event condition. Therefore, both the *push newest* and the *event* pattern provide an interface based on member functions only. A handler based approach is, however, used by the *event* pattern for the implementation of the event condition and by the *push timed* pattern to handle the timer event indicating that new data has to be acquired for distribution.

The user access modes are all thread safe and can be called from concurrent tasks without further synchronization. This already moves a large source of errors and pitfalls in multithreaded programs from the user scope into the responsibility of the framework.

Further analysis of the assignment of user access modes to communication patterns shows that blocking member functions that spend significant time in the communication pattern occur only at the service requestors. Method invocatios at the service providers are only used to provide calculated results or to provide new data to be distributed to subscribed clients. These member function calls are handled in very short time. In contrast, member functions at service requestors wait for an answer or for the next arriving update. Since this may take an arbitrary long time, the service requestors provide a blocking flag, which can be set to abort all blocking calls and to prevent subsequent calls from blocking. This allows graceful deactivation of pending services when waiting is not appropriate anymore.

5.4.5 Naming of Services

Communication patterns always provide a link between a service requestor and a service provider. Although they can even be used to implement interfaces inside a component, both parts are normally located not only in different components, but the components are also distributed over different hosts. One therefore needs a scheme to locate and to address a specific service within a specific component.

Figure 5.21 illustrates the chosen approach. Component interfaces are tagged with user definable names for clear and simple references. A name based approach is easy to use and allows intuitive

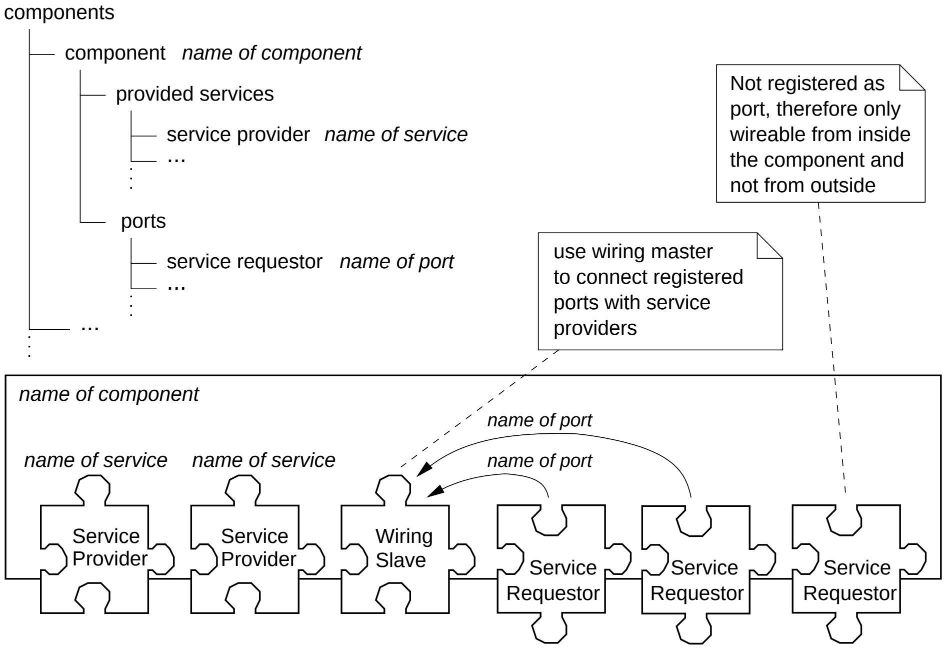

components
component name of component
provided services
service provider name of service
...
ports
service requestor name of port
...
...
Not registered as port, therefore only wireable from inside the component and not from outside
use wiring master to connect registered ports with service providers
name of component
name of port
name of port
name of service
name of service
Service Provider
Service Provider
Wiring Slave
Service Requestor
Service Requestor
Service Requestor

Figure 5.21: *Naming of components and services.*

service designators. This results in ad-hoc understandable descriptions of component wirings.

Each component has a unique name. Services provided by a component are also denoted by names that are unique within a component. Service requestors do not have public names since they are not contacted as it is the case with service providers. Since links between service requestors and service providers are always initiated from the service requestor, the latter forwards a reference to itself to be henceforth accessible from the service provider without requiring a public identifier. Service requestors can register themselves at the wiring service of their component to become named ports. As named ports, service requestors can be wired with service providers from outside the component. Port names also have to be unique within a component.

Services can now be clearly identified by a tuple $\{$*component, service*$\}$ composed of the name of the service providing component and the name of the service. A service requestor only needs to know this tuple to connect to a service provider. The wiring pattern can now be based on fourtuples $\{$*A:component, B:port, C:component, D:service*$\}$ to connect the service requestor port *B* of component *A* with the service provider *D* of component *C*. The framework, of course, only establishes connections to services which are compatible with the requestor.

5.4.6 The Overall Picture

The overall picture of the approach is shown in figure 5.22. All component interfaces are composed of standardized patterns which transmit communication objects. Components can be assembled to form complex applications due to standardized interface patterns and can even be wired dynamically at runtime from inside and outside a component. Communication patterns separate the communication from the application architecture, follow the design goal of decoupling and provide transparency aligned to the needs of robotic applications.

Table 5.3 summarizes the externally visible interface of each component shown in figure 5.22.

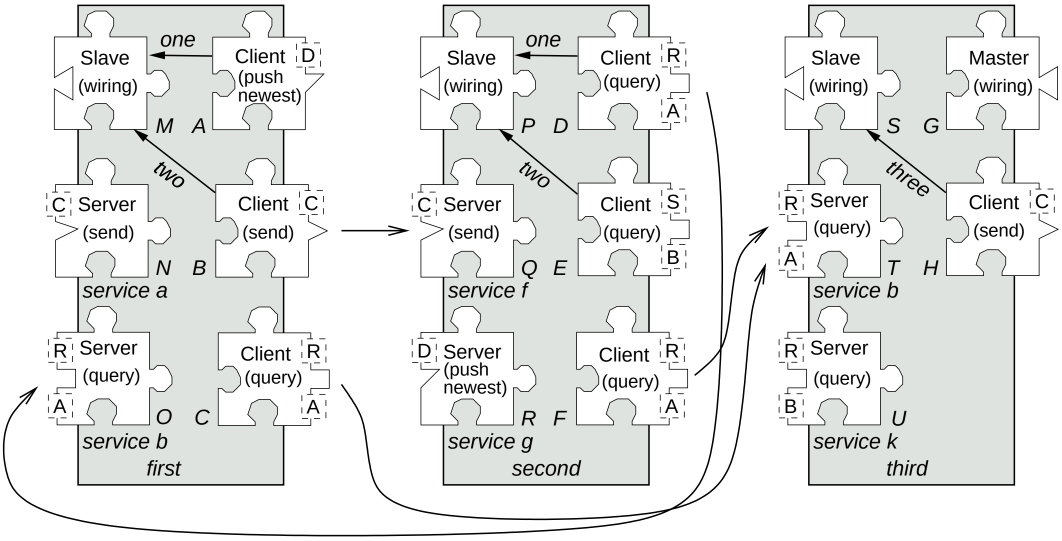


__Figure 5.22:__ The overall picture of the service based component approach.

first component		**second component**		**third component**	
used service	port	**used service**	port	**used service**	port
A : push newest<D> *B* : send<C> *C* : query<R,A>	*one* *two* -	*D* : query<R,A> *E* : query<S,B> *F* : query<R,A>	*one* *two* -	*G* : wiring *H* : send<C>	- *three*
provided service	name	**provided service**	name	**provided service**	name
M : wiring *N* : send<C> *O* : query<R,A>	- *service a* *service b*	*P* : wiring *Q* : send<C> *R* : push newest<D>	- *service f* *service g*	*S* : wiring *T* : query<R,A> *U* : query<R,B>	- *service b* *service k*

__Table 5.3:__ Summary of the externally visible component interfaces of the example of figure 5.22.

The interfaces *A*, *B*, *D*, *E* and *H* expose themselves as ports and are thus wireable from outside a component. The wiring master *G* can access all three components via their wiring slaves *M*, *P* and *S*. In principle, one can connect *A* with *R* and one can connect *B* with *N* and *Q* and *H* with *N* and *Q*. Furthermore, *C*, *D* and *F* can be connected with *O* and *T* and there is no appropriate service available for *E*. The wirings configurable from outside a component via the ports are summarized in table 5.4.

The level of transparency provided by the communication patterns is just such that one can still control and adjust major aspects of components in distributed systems without abandoning the amenities of transparency. *Access* transparency, for example, hides differences in data representations. That is implemented within communication objects by standardized descriptions of data structures to be transmitted. *Location* transparency is provided by the naming service used to identify services. Accessing a component on a different host is transparent but due to the fact that component external interactions are always based on communication patterns, a component builder is always fully aware of when communication occurs. This is further ensured by always transmitting communication objects *by value*. Therefore, using communication objects within a component never results in unexpected re-

$\{component, port\}$	$\{component, service\}$
$A : \{first, one\}$	$R : \{second, service\ g\}$
$B : \{first, two\}$	$N : \{first, service\ a\}, Q : \{second, service\ f\}$
$D : \{second, one\}$	$O : \{first, service\ b\}, T : \{third, service\ b\}$
$E : \{second, two\}$	–
$H : \{third, three\}$	$N : \{first, service\ a\}, Q : \{second, service\ f\}$

Table 5.4: *Summary of the wirings of the example shown in figure 5.22 settable from outside a component by a wiring master.*

mote object access which causes unforeseen and large delays. Compared to this, when using standard distributed object approaches, it is very hard for component builders to be always fully aware of the total amount of communication involved in a simple method call. The presented approach, therefore, greatly simplifies the predictability of the overall timing behavior of a component. *Migration* transparency is given because services are always only connected by names irrespective of where they are located. By purpose, the designator of a service comprises not only the name of the service but also the name of the service providing component. Connecting to a service, therefore, always clarifies which component is involved and knowing where that component is located gives a rough estimate of the expected delay in communication. The dynamic wiring of components even allows transparent access to compatible services in different components located on arbitrary hosts. Controlling the wirings, however, allows the application builder to tune the overall system performance by using colocated services for critical interactions. The approach does not support *relocation* transparency which would hide the fact that a resource has moved to another location while in use. This would make it nearly impossible for application builders to predict the expected delays in communication when plugging together an application. The approach, however, provides transparency with respect to *concurrency* which is hiding that a resource is shared by several competitive users. The communication patterns support concurrent access not only from inside a component but also at the component level where any number of clients can access a server. The user interfaces of the patterns are defined such that the component builder can decide on both the processing model and the provided resources. For example, one can attach a thread pool, a single thread with a processing queue or even a passive handler only. This again is important to achieve predictability which is difficult if the resources are assigned behind the scenes of a framework.

5.5 The Component Builder View on the Approach

Implementing a framework based on the proposed approach requires a precise specification of the communication patterns and their access modes. The specification can be split into aspects that are relevant to the framework builder only and those that define the user interface and that are thus also relevant to the framework users. Aspects that are only relevant to the framework builder concern the pattern internal communication across components and the pattern internal organization of the various user access modes. These are described in section 5.6. In contrary, the component builder focuses on the semantics of the access modes of the communication patterns and aspects of the overall usage of components. Generally, means to describe the details of the proposed approach range from formal methods to concrete implementations. Using formal methods would result in hard to read and unnecessarily complex specifications that completely ignore software technical aspects. Implementations, on the other hand, get lost in details which can be solved in an arbitrary manner and miss the abstraction to get the principles behind an approach. Nevertheless, many aspects are best illustrated based on a concrete implementation. The user view, therefore, uses *C++* notions wherever this significantly simplifies the description. This, however, does not mean that the presented concepts cannot be implemented in any other object oriented language. An overview on the core patterns of the SMARTSOFT framework and their interactions is given in figure 5.23.

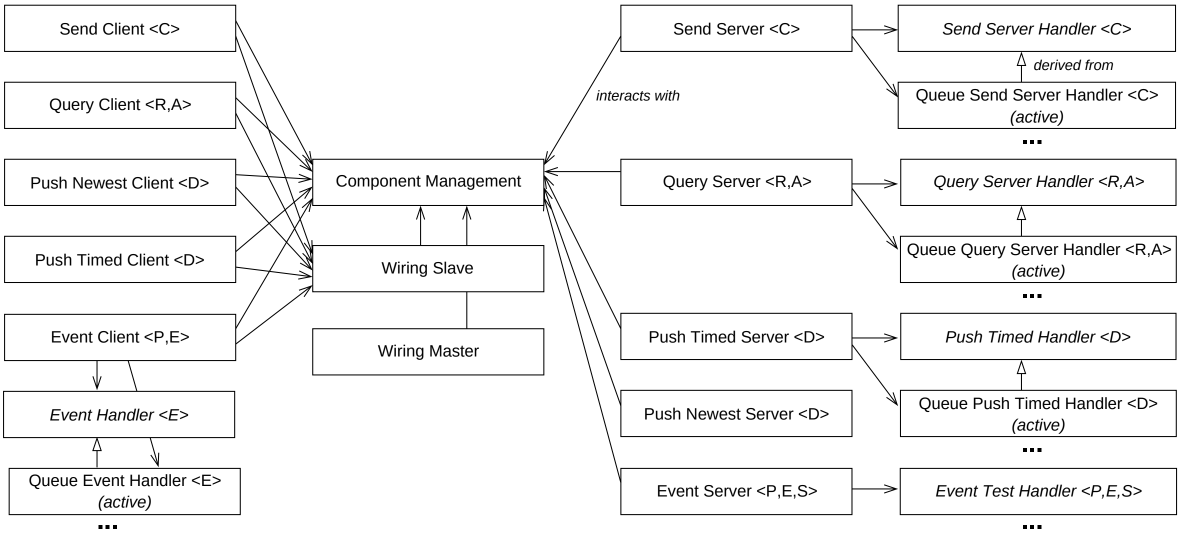


Figure 5.23: *The core patterns of the* SMARTSOFT *framework. Active handlers are provided for those patterns where use cases indicate time intensive computations in handlers. Italic class names indicate abstract base classes.*

5.5.1 The Component Management

A central role within every component is played by the component management class shown in figure 5.24. Each component possesses exactly one instance of this class, which wraps the basic framework infrastructure. The constructor of the component management class requires the name of the component which is subsequently used to address this component by name.

Component Management

```
+ SmartComponent(name:const string&,argc:int&,argv:char**) throw(SmartError)
+ ~SmartComponent() throw() [virtual]

+ blocking(flag:const bool) : StatusCode throw()          // blocking mode effective component wide

+ run() : StatusCode throw()                               // operates framework activities
```

Figure 5.24: *The class diagram of the component management class.*

The framework is normally operated by the main thread of a component which calls the *run* member function. The framework activities are completely independent of the user activities and keep the component external communication alive. All user activities within a component are moved to separate threads as shown in figure 5.25.

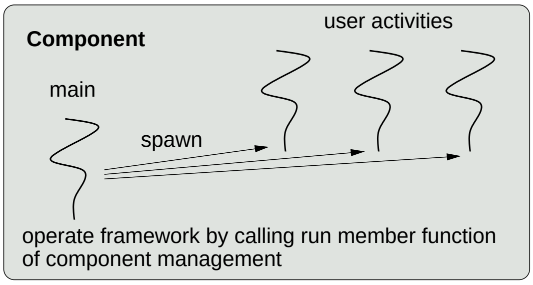


Figure 5.25: *User activities are separated from the framework.*

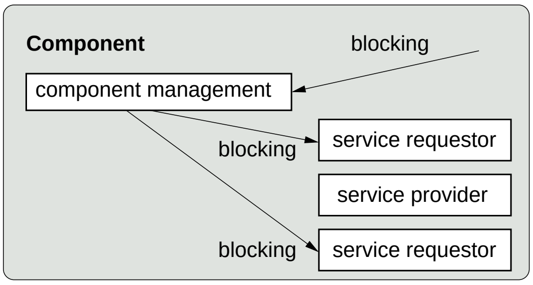
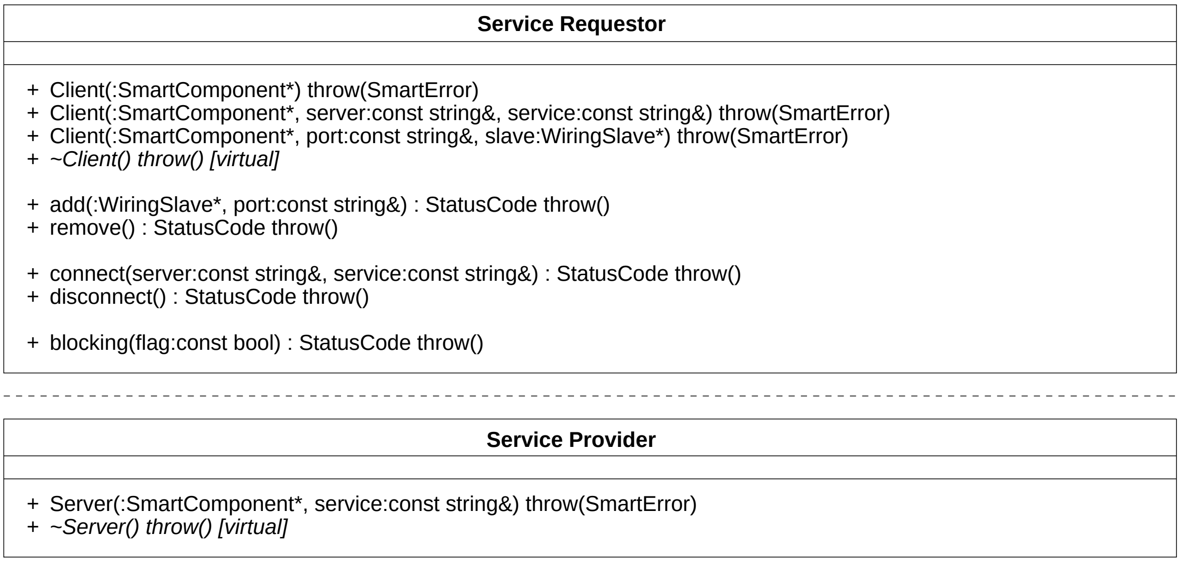


Figure 5.26: *The blocking mode of the component management which is effective component wide.*

Furthermore, the component management class manages a blocking mode indicator which is effective component wide as illustrated in figure 5.26. If the blocking mode is changed from *true* to *false*, all blocking member functions of all services of a component are properly discarded and any subsequent calls to blocking member functions return immediately with an appropriate status code. This is of great use for graceful deactivation of user activities within a component and is used by state management mechanisms. The component management class is the obvious place to handle the component wide blocking mode since there anyway is a close interaction of the component management with every service requestor and service provider of a component.

5.5.2 The Communication Patterns

Each communication pattern consists of a service provider and a service requestor. Figure 5.27 shows the user interfaces common to all communication patterns except the *wiring* pattern.



Service Requestor

```
+ Client(:SmartComponent*) throw(SmartError)
+ Client(:SmartComponent*, server:const string&, service:const string&) throw(SmartError)
+ Client(:SmartComponent*, port:const string&, slave:WiringSlave*) throw(SmartError)
+ ~Client() throw() [virtual]

+ add(:WiringSlave*, port:const string&) : StatusCode throw()
+ remove() : StatusCode throw()

+ connect(server:const string&, service:const string&) : StatusCode throw()
+ disconnect() : StatusCode throw()

+ blocking(flag:const bool) : StatusCode throw()
```

Service Provider

```
+ Server(:SmartComponent*, service:const string&) throw(SmartError)
+ ~Server() throw() [virtual]
```

Figure 5.27: *The user API common to all communication patterns except the* wiring *pattern.*

The constructor of a *service provider* always requires the specification of the name of the offered service. Constructors of *service requestors* always offer three different modes. The first option is to create an instance not connected to a service provider. The second option is to connect to the service specified by component name and service name. The constructor blocks until the requested service gets available and until the connection is established. The constructor fails if the chosen service provider is incompatible. The last option is to expose the service requestor instance as port wireable from other components. The constructor expects a port name which can later on be used by the wiring pattern to connect this port with an appropriate service provider. The constructor used to create a service requestor instance does not determine any future mode of a service requestor. A service requestor can decide on being wireable from outside the component at any point of time using the *add/remove* member functions. Likewise, the connection to a service provider can be changed at any point of time using the *connect/disconnect* member functions.

connect	Connect this service requestor to the denoted service provider. An already established connection is first disconnected.
ok	Connected to the specified service provider.
service unavailable	The specified service provider is not available and the requested connection cannot be established. The service requestor is not connected to any service provider.
service incompatible	The specified service provider is not compatible to the service requestor and can therefore not be connected. The service requestor is not connected to any service provider.
communication error	Something went wrong at the level of the intercomponent communication and the connection is not established. The service requestor is not connected to any service requestor.
error	Something went wrong and no connection is established. The service requestor is not connected to any service provider.

disconnect	Disconnects the service requestor from the service provider. A disconnect always aborts and properly cleans up any pending communication.
ok	Disconnected from the service provider.
communication error	Something went wrong at the level of the intercomponent communication. At least the service requestor part is in the disconnected state independently of eventually not properly executed clean up procedures at the service provider.
error	Something went wrong. At least the service requestor part is in the disconnected state independently of eventually not properly executed clean up procedures at the service provider.

Table 5.5: *The connect/disconnect member functions to change the wiring at runtime from inside the component.*

Table 5.5 explains the semantics of the *connect/disconnect* member functions of service requestors. The *connect* member function expects a tuple $\{$*component, service*$\}$ which denotes the component and the service to which the service requestor is to be connected to. It then checks whether the service provider is compatible in terms of the communication pattern and the communication objects. The *disconnect* member function disconnects the service requestor from the service provider. The patterns have built-in mechanisms to allow changes to be made to the link between a service requestor and a service provider at any point of time without further synchronization at the user level. A disconnect for example correctly aborts blocking calls which would otherwise keep waiting for messages not arriving anymore after a disconnect.

The *add/remove* member functions are summarized in table 5.6. Generally, the *add* member function exposes a service requestor as port via the wiring pattern. The wiring pattern can establish connections from outside a component. The *add* member function expects a port name and adds the service requestor to the wiring slave of the component. The service requestor gets wirable from outside via an externally visible port. This enables an external wiring master to connect that service requestor via its port name with appropriate service providers. The *remove* member function removes

add	Add this service requestor to the set of ports wireable via the wiring pattern from outside the component. Already established connections keep valid. If this service requestor is already exposed as port, it is first removed and then added with the new port name.
ok	Service requestor added to the set of ports.
port already used	Port name already in use and thus this service requestor now not available as port with that name.
no wiring slave	No wiring slave provided with object creation and service requestor cannot be exposed as port.
error	Something went wrong and this service requestor is not available as port.

remove	Remove this service requestor from the set of ports wireable via the wiring pattern from outside the component. Already established connections keep valid but can now be changed only from inside and not from outside this component anymore.
ok	Service requestor not exposed as port anymore (or was not registered as port or no wiring slave available).
error	Something went wrong but the service requestor is removed from the set of ports in any case.

Table 5.6: *The add/remove member functions to expose a service requestor as wireable port.*

a previously added service requestor from the wiring slave. Again, the patterns take care that one can add or remove a port at any point of time without further synchronization at the user level.

blocking	Allow blocking calls or abort and reject blocking calls.
ok	Changed the blocking mode of the service requestor.
error	Something went wrong.

Table 5.7: *The blocking member function to discard blocking member function calls.*

Furthermore, each service requestor also provides a *blocking* member function to set an internal state indicating whether blocking is allowed. A service requestor blocks in member functions calls only if both the service requestor and the component wide indicator allow blocking. If blocking is set to *false* then already blocking calls are aborted and subsequent calls return immediately. In both cases, the returned status code is set to *cancelled*. Again, no further synchronization is needed at the user level. The *blocking* member function provides the basis for ordered suspension of communication activities. This is needed for graceful deactivation of component activities and is the basis for a component state management.

5.5.2.1 The Send Pattern

The *send* pattern implements a one-way communication. It provides a member function based interface at the service requestor and a handler based interface at the service provider as shown in the class diagrams in figure 5.28.

The service requestor can send communication objects to the service provider using the *send* member function shown in detail in table 5.8. At the service provider, each incoming communication object is forwarded to the registered handler for further processing. The service requestor and the service provider of the send pattern are applied by binding the templates with a communication object. The service provider additionally requires a handler that has to be derived from the abstract handler class to provide an implementation of the *handleSend* member function which processes the incoming commands.

send	Perform an asynchronous one-way communication.
ok	Everything is ok and communication object sent to server.
disconnected	The client is disconnected and no send can be made.
communication error	Communication problems, data not transmitted.
error	Something went completely wrong.

Table 5.8: *The send member function to perform a one-way communication.*

Figure 5.29 shows the sequence diagram of both the client and the server side behavior. Due to the used passive handler, the execution of the second received command is delayed until the first command is finished. The *Queue Send Server Handler* is a decorator for the handler class to execute the *handleSend* member function in a separate thread. Its *handleSend* member function enqueues all commands in a queue. From there, they are processed by the thread of the active handler independently of the communication activity by calling the original *handleSend* member function.

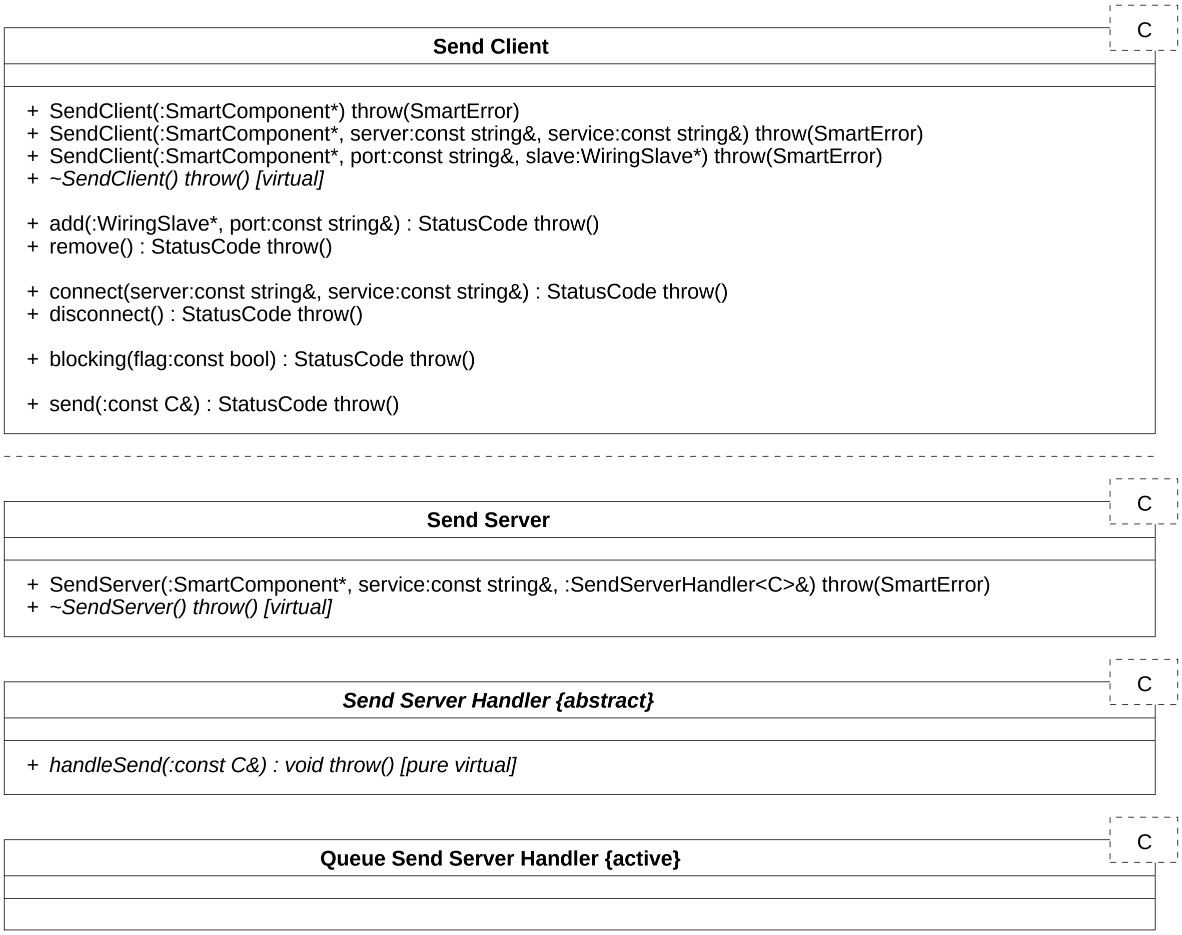

C
Send Client
+ SendClient(:SmartComponent*) throw(SmartError)
+ SendClient(:SmartComponent*, server:const string&, service:const string&) throw(SmartError)
+ SendClient(:SmartComponent*, port:const string&, slave:WiringSlave*) throw(SmartError)
+ ~SendClient() throw() [virtual]
+ add(:WiringSlave*, port:const string&) : StatusCode throw()
+ remove() : StatusCode throw()
+ connect(server:const string&, service:const string&) : StatusCode throw()
+ disconnect() : StatusCode throw()
+ blocking(flag:const bool) : StatusCode throw()
+ send(:const C&) : StatusCode throw()
C
Send Server
+ SendServer(:SmartComponent*, service:const string&, :SendServerHandler<C>&) throw(SmartError)
+ ~SendServer() throw() [virtual]
C
Send Server Handler {abstract}
+ handleSend(:const C&) : void throw() [pure virtual]
C
Queue Send Server Handler {active}

Figure 5.28: *The class diagrams of the send pattern.*

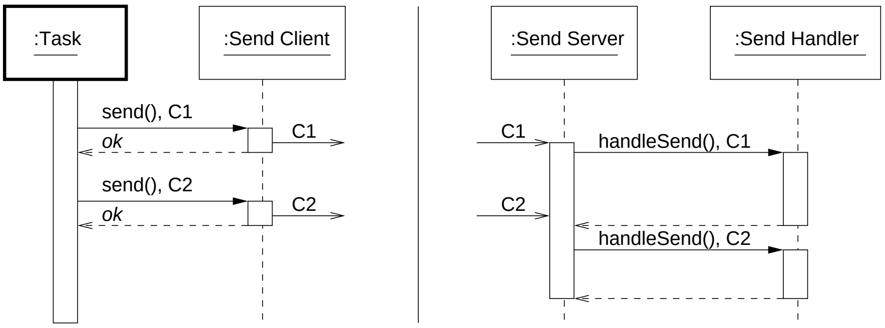


Figure 5.29: *The client and server side behavior of the send pattern.*

5.5.2.2 The Query Pattern

The *query* pattern implements a two-way communication. It provides a member function based interface at the service requestor and a handler based interface at the service provider as shown in figure 5.30. The templates require two communication objects, one for the request and one for the answer.

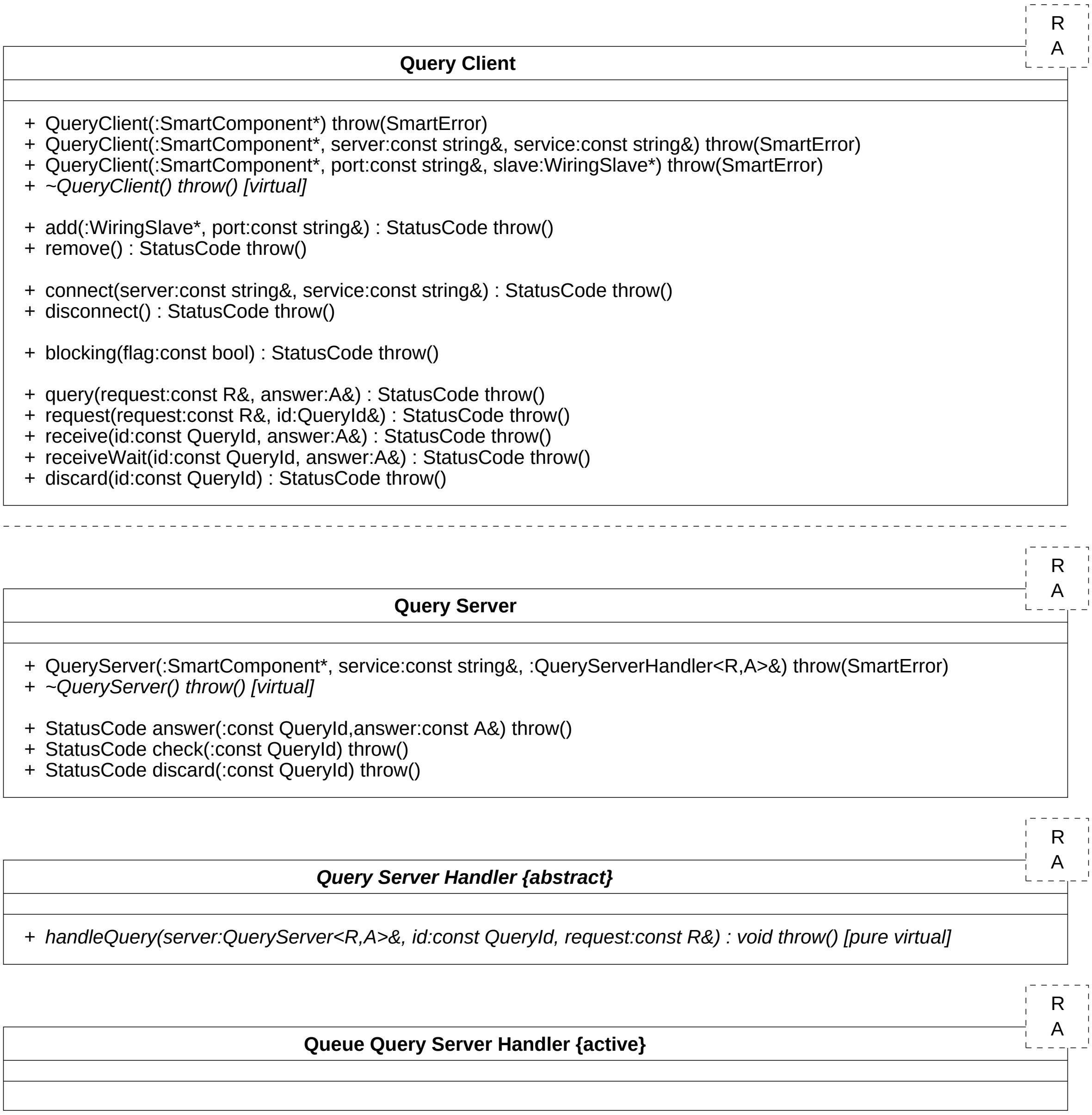

R
A
Query Client
+ QueryClient(:SmartComponent*) throw(SmartError)
+ QueryClient(:SmartComponent*, server:const string&, service:const string&) throw(SmartError)
+ QueryClient(:SmartComponent*, port:const string&, slave:WiringSlave*) throw(SmartError)
+ ~QueryClient() throw() [virtual]
+ add(:WiringSlave*, port:const string&) : StatusCode throw()
+ remove() : StatusCode throw()
+ connect(server:const string&, service:const string&) : StatusCode throw()
+ disconnect() : StatusCode throw()
+ blocking(flag:const bool) : StatusCode throw()
+ query(request:const R&, answer:A&) : StatusCode throw()
+ request(request:const R&, id:QueryId&) : StatusCode throw()
+ receive(id:const QueryId, answer:A&) : StatusCode throw()
+ receiveWait(id:const QueryId, answer:A&) : StatusCode throw()
+ discard(id:const QueryId) : StatusCode throw()
R
A
Query Server
+ QueryServer(:SmartComponent*, service:const string&, :QueryServerHandler<R,A>&) throw(SmartError)
+ ~QueryServer() throw() [virtual]
+ StatusCode answer(:const QueryId,answer:const A&) throw()
+ StatusCode check(:const QueryId) throw()
+ StatusCode discard(:const QueryId) throw()
R
A
Query Server Handler {abstract}
+ handleQuery(server:QueryServer<R,A>&, id:const QueryId, request:const R&) : void throw() [pure virtual]
R
A
Queue Query Server Handler {active}

Figure 5.30: *The class diagrams of the query pattern.*

The Client Side The client interface provides two different access modes. The first one is the standard blocking query shown in table 5.9 which forms a synchronous interface. The second access mode forms an asynchronous interface and supports the deferred reception of answers for previously invoked requests and is shown in tables 5.10, 5.11, 5.12 and 5.13. That access mode provides both blocking and non-blocking member functions. It is particularly useful for interleaved requests of different queries invoked at different query clients connected to different service providers which then calculate the responses in parallel.

query	Perform a synchronous blocking query that is invoke the request and await the response.
ok	Everything is ok and *answer* contains the answer.
cancelled	Blocking is not allowed or is not allowed anymore and, therefore, the pending query is aborted, the answer is lost and no valid answer is returned.
disconnected	The client is either disconnected and no query can be made or it got disconnected and a pending query is aborted without answer. In both cases, *answer* is not valid.
wrong identifier	The query got discarded at the service provider and thus, no valid answer is returned.
communication error	Communication problems, *answer* not valid.
error	Something went completely wrong and *answer* is not valid.

Table 5.9: *The query member function to perform a two-way communication.*

request	Invoke a query and receive the answer later (asynchronous).
ok	Everything is ok and *id* is a valid query identifier used to either fetch or discard the answer.
disconnected	Request is rejected since client is not connected to a server and therefore *id* is not a valid identifier.
communication error	Communication problems, *id* is not valid.
error	Something went completely wrong, *id* is not valid.

Table 5.10: *Invoking a request with deferred answer at the service requestor.*

receive	Non-blocking call to fetch the answer belonging to the given identifier (asynchronous).
ok	Everything is ok and *answer* contains the answer. The *id* is consumed and is not valid any longer.
no data	The answer is not yet available, therefore try again later. The *id* keeps valid but *answer* contains no answer.
disconnected	The answer belonging to the identifier cannot be received anymore since the client got disconnected. *Id* is not valid any longer and no valid *answer* is returned.
wrong identifier	No pending query with this identifier available, therefore no valid *answer* returned. The *id* either was already invalid or got invalid. It gets invalid by consuming it, either by a concurrent call to the *receive* or the *receive wait* method with the same *id* or by a *discard*, either at the client or at the server.
error	Something went completely wrong, *id* is not valid any longer and *answer* contains no answer.

Table 5.11: *Getting a deferred answer at the service requestor.*

Every request is assigned a unique identifier which is later on used to fetch the correct answer. An identifier stays valid as long as it is not consumed. Although it is permitted, it makes no sense to use the same identifier in concurrent method calls since only one invocation can consume the identifier and all others experience a *wrong identifier* status. An identifier gets consumed by the *receive* and the *receive wait* methods either by returning a valid answer or by reporting a *disconnected* state. A query identifier can also be consumed by a *discard*, either called at the service requestor or at the service provider. The *receive* and the *receive wait* method then return a *wrong identifier* status since

receiveWait	Blocking call to fetch the answer belonging to the given identifier (asynchronous).
ok	Everything is ok and *answer* contains the answer.
cancelled	Blocking is not allowed or is not allowed anymore and therefore blocking call is aborted and no valid *answer* is returned. The query identifier keeps valid.
disconnected	The answer belonging to the identifier cannot be received anymore since the client got disconnected. *Id* is not valid any longer and no valid *answer* is returned.
wrong identifier	No pending query with this identifier available, therefore no valid *answer* returned. The query identifier either was already invalid or got invalid (see *receive* method).
error	Something went completely wrong, *id* is not valid any longer and *answer* contains no answer.

Table 5.12: *Waiting for a deferred answer at the service requestor.*

discard	Discard the deferred answer with the given identifier and invalidate the identifier. Call this member function if you do not want to get the answer of a request anymore which was invoked by *request*. Aborts a blocking *receive wait* and also informs the service provider about the discarded request.
ok	Everything is ok and the pending query with identifier *id* is discarded.
wrong identifier	No pending query with this identifier available.
error	Something went completely wrong and *id* is not valid any longer.

Table 5.13: *Discard a deferred answer at the service requestor.*

the identifier in use got consumed. In case the identifier in use gets consumed, a blocking *receive wait* method gets unblocked.

Even though the query identifier used inside the *query* method is never returned to the user level and can thus not be used by any client side method, the *query* method can return a *wrong identifier* status since the request can get discarded at the service provider. The *wrong identifier* status indicates that the request either got rejected at the service provider before it got forwarded to the request handler or got discarded by calling the *discard* method at the service provider. In case of using the *request* method to invoke a query, the identifier can get discarded also by the client side *discard* method.

A query identifier is not consumed by a *disconnect* and answers already received but not yet fetched are not affected by a disconnect. Disconnecting a query client not only correctly sets the status of still open answers and aborts blocking calls with the appropriate status code but a disconnect also cleans up the list of open requests at the service provider to make sure that answers provided for meanwhile disconnected clients are discarded.

The status codes are organized such that always only the most important reason is returned. For example, one first has to possess a valid identifier before one can get more details on the state of a request. Thus, an invalid identifier is the most important reason and being disconnected is more important than the state of the blocking mode since the blocking mode is of no interest as soon as one cannot get the answer anymore. Thus, the *wrong identifier* status is returned in case the used identifier is not valid or in case it got invalid. The *disconnected* status is returned as soon as there is no server connection available irrespective of the blocking mode. The blocking mode is the third most important reason for refusing a blocking call and therefore *cancelled* is returned if the identifier is valid and the client is connected to a server but blocking is not allowed anymore.

Figure 5.31 summarizes the behavior of the client side of the query pattern. A client can handle any number of concurrent and nested queries. Figure 5.32 shows the effect of changing the blocking mode and figure 5.33 shows a disconnect performed during active queries.

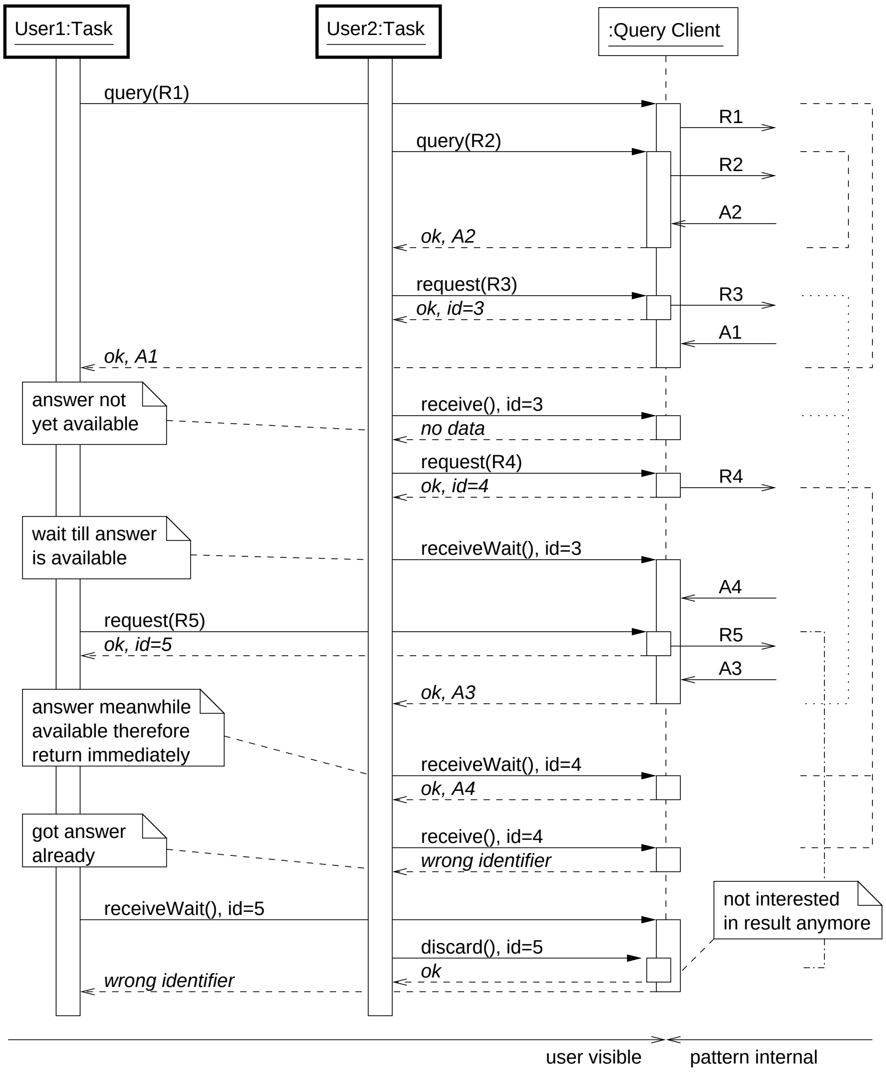


Figure 5.31: *Sequence diagram of the concurrent and interleaved use of the client side user interface of the query pattern.*

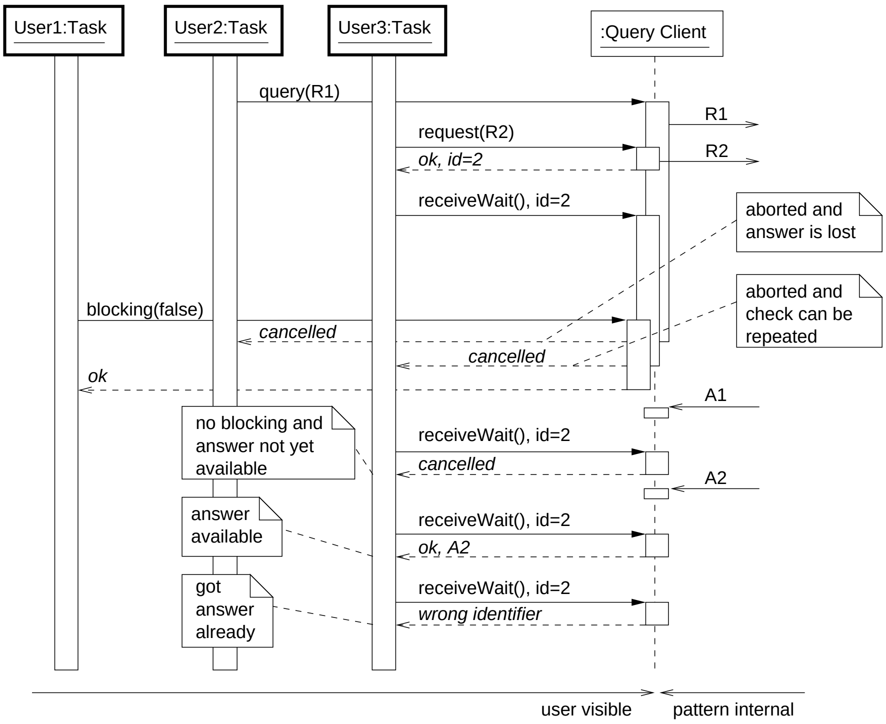


Figure 5.32: *The client side behavior of the query pattern when blocking is set to* false.

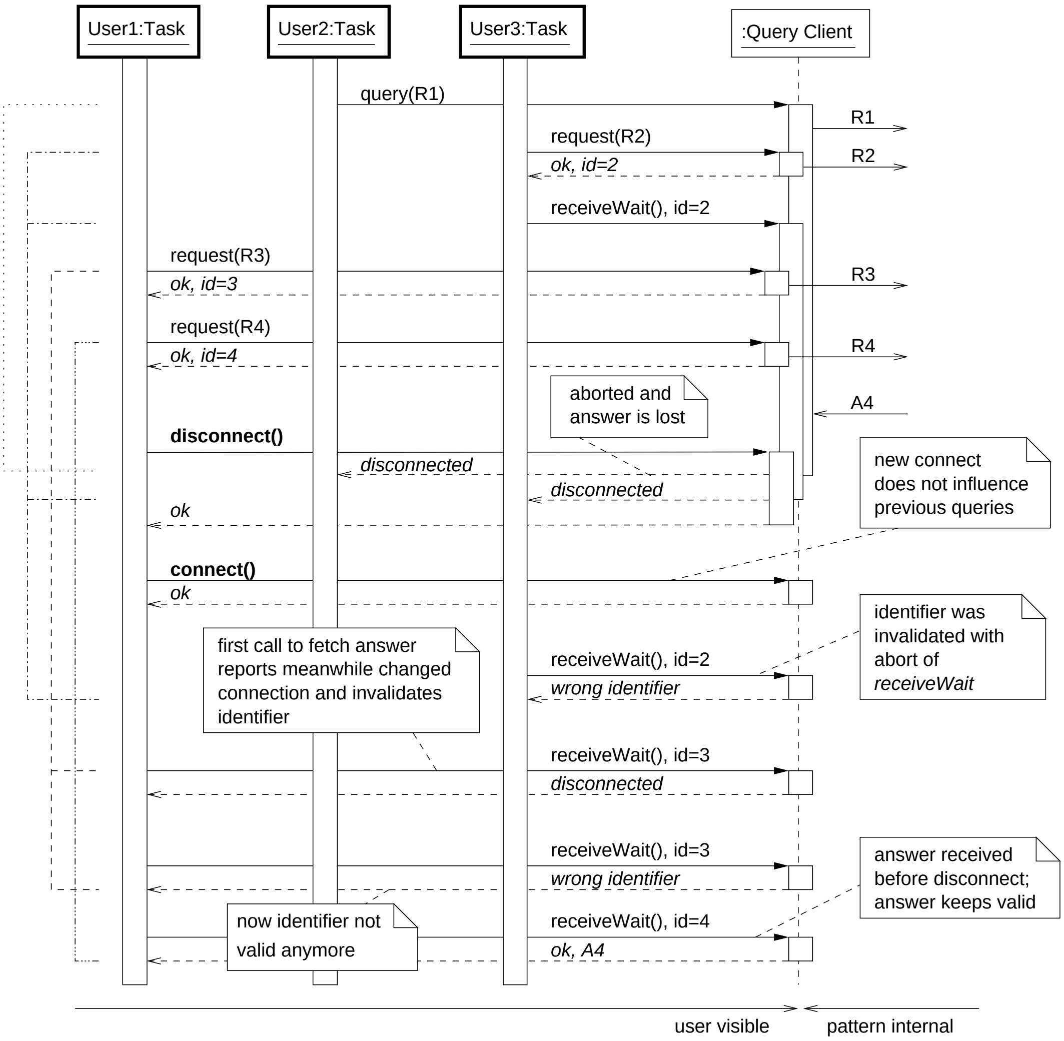


Figure 5.33: *The client side behavior of the query pattern when client is disconnected and reconnected to another service provider.*

The Server Side The server provides a handler based interface to which every incoming request is forwarded to. The abstract handler class requires to implement the *handle query* member function for processing incoming requests. An identifier provided at the handler with every request has to be passed on with the answer to non-ambiguously assign answers to open requests and to properly send back the answer inside the pattern. The server makes no assumptions on the order of the answers and it is up to the user of the pattern to implement any kind of processing mechanism such as prioritized queues or even a processing pipeline. The separate answer member function makes it much simpler to implement different processing models than it would be the case with a handler returning the answer. The handler can again be decorated to become an active handler which executes the *handle query* member function in a separate thread and manages requests via a queue, for example.

answer	Provide answer to be sent back to the requestor.
ok	Everything ok and answer was sent to requesting client. The identifier is not valid any longer.
wrong identifier	No pending query with that identifier known. The *id* either never has been valid, the answer was already provided or that request got discarded.
disconnected	Answer not needed anymore since requesting client got disconnected meanwhile. The answer is discarded and the identifier is not valid any longer.
communication error	Communication problems.
error	Something went completely wrong.

Table 5.14: *Provide the answer at the query service provider.*

check	Check whether the identifier still belongs to a request that needs to be answered.
ok	Identifier is still valid and request still needs to be processed since response is expected.
wrong identifier	No pending query with that identifier known. The *id* either never has been valid, the answer was already provided or that request got discarded. In any case, the processing of the enquired *id* need not to be continued.
disconnected	Answer not needed anymore since requesting client got disconnected meanwhile. The identifier is not valid any longer.
error	Something went completely wrong.

Table 5.15: *Check at the service provider whether request still has to be processed.*

discard	Discard that request and invalidate the identifier. Call this member function if the request has to be discarded for any reason.
ok	Successfully discarded request and also informed client about the discarded request in case the identifier belonged to a still to be processed request. The identifier is not valid anymore.
wrong identifier	No pending query with that identifier known (see *check* method).
error	Something went completely wrong.

Table 5.16: *Discard request at the service provider.*

Figure 5.14 shows the *answer* method to return a response. Each request can be answered only once and the response gets discarded in case the client got disconnected meanwhile or in case the client discarded the request. Due to the handler based interface, there is no simple way to abort the processing of no longer needed requests. All requests are immediately forwarded to the handler and are never stored in the service provider. Thus, the *check* method shown in figure 5.15 allows to check whether a request still needs to be processed. With the *discard* method shown in table 5.16, one can discard a request in case the service provider is overloaded, for example. The *discard* properly informs the service requestor so that, for example, all blocking calls get properly unblocked. The server side behavior of the query pattern is summarized in figures 5.34 and 5.35.

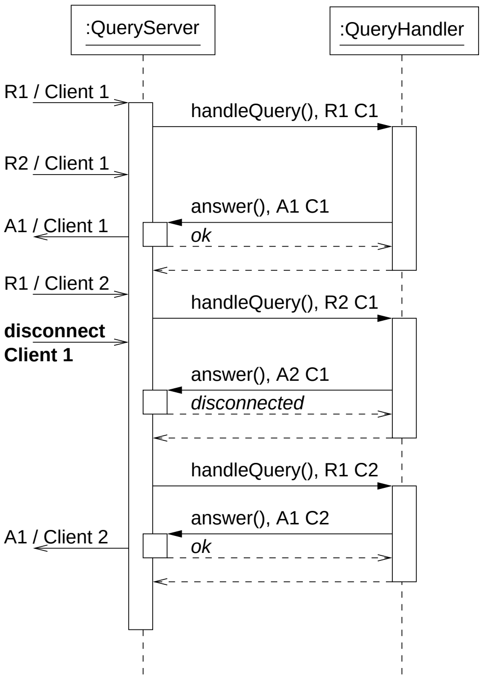


Figure 5.34: *The server side behavior of the query pattern with a passive handler.*

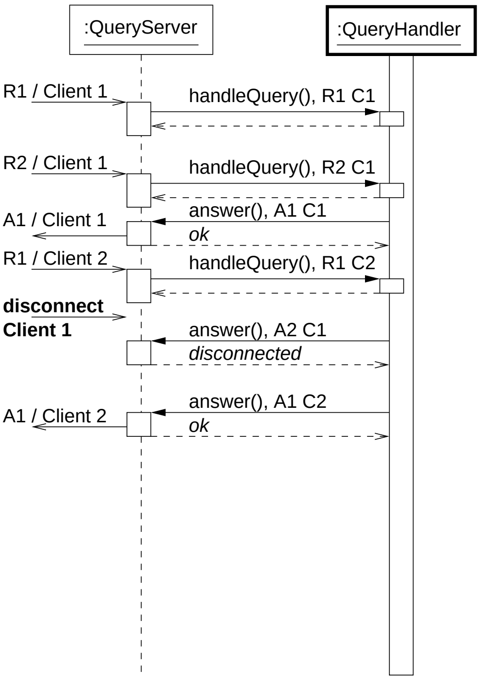
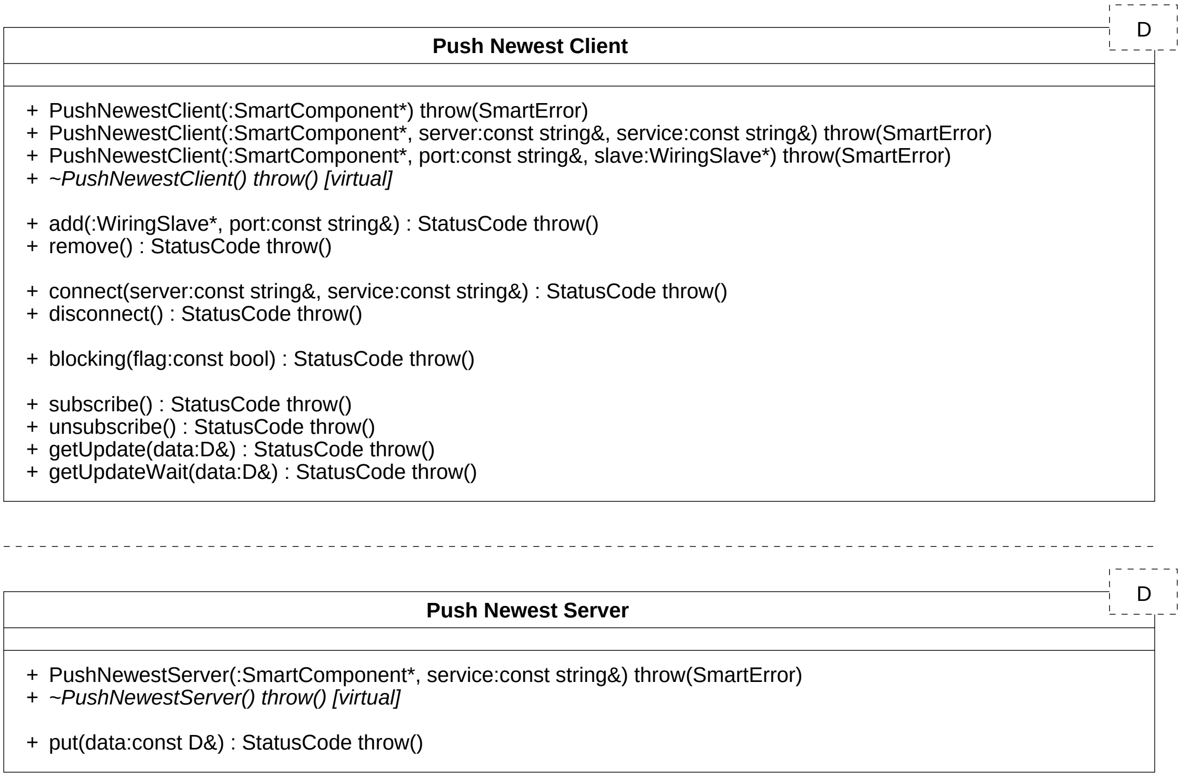


Figure 5.35: *The server side behavior of the query pattern with an active handler.*

5.5.2.3 The Push Newest Pattern

The *push newest* pattern provides a member function based interface at both the service provider and the service requestor as shown in figure 5.36.



```
+ PushNewestClient(:SmartComponent*) throw(SmartError)
+ PushNewestClient(:SmartComponent*, server:const string&, service:const string&) throw(SmartError)
+ PushNewestClient(:SmartComponent*, port:const string&, slave:WiringSlave*) throw(SmartError)
+ ~PushNewestClient() throw() [virtual]

+ add(:WiringSlave*, port:const string&) : StatusCode throw()
+ remove() : StatusCode throw()

+ connect(server:const string&, service:const string&) : StatusCode throw()
+ disconnect() : StatusCode throw()

+ blocking(flag:const bool) : StatusCode throw()

+ subscribe() : StatusCode throw()
+ unsubscribe() : StatusCode throw()
+ getUpdate(data:D&) : StatusCode throw()
+ getUpdateWait(data:D&) : StatusCode throw()
```

```
+ PushNewestServer(:SmartComponent*, service:const string&) throw(SmartError)
+ ~PushNewestServer() throw() [virtual]

+ put(data:const D&) : StatusCode throw()
```

Figure 5.36: *The class diagrams of the push newest pattern.*

The Client Side A client needs to be subscribed to a service provider to get updated as soon as new data is available at the server. A client can, of course, only get subscribed if it is connected to a service provider. A disconnect automatically performs an unsubscribe. Once subscribed, a client always holds the latest data from the service provider. A client gets its first data with the first update after subscription and not with subscribing. This prevents the client from holding data which has been calculated before subscription. This, of course, can result in quite a long time before the first data is available at the client. If this is undesired for a service, it is the responsibility of the component builder to eventually distribute the same data several times. This avoids that new subscriptions are kept uninformed for a too long time.

A client can either get the currently available latest data by calling *getUpdate* or it can explicitly wait for the next update by calling *getUpdateWait*. Again, only the most important reason for data not being available is returned. Being disconnected is more important than being unsubscribed and that is more important than being cancelled. The client side behavior of the push newest pattern is summarized in figure 5.37.

subscribe	Subscribe at the server to get updates as soon as new data is available there.
ok	Everything is ok and client is subscribed.
disconnected	Client is not connected to a server and can therefore not subscribe for updates. Client is still unsubscribed.
communication error	Communication problems and client is not subscribed.
error	Something went completely wrong and client is not subscribed.

Table 5.17: *Subscribe for updates at the client side.*

unsubscribe	Unsubscribe to get no more updates.
ok	Everything is ok and client is now unsubscribed or client has already been unsubscribed. All blocking calls are aborted with the appropriate status and yet received and still buffered data is deleted to avoid returning old data.
communication error	Communication problems and client is not unsubscribed.
error	Something went completely wrong and client is not unsubscribed.

Table 5.18: *Unsubscribe at the client side to get no more updates.*

getUpdate	Non-blocking call to immediately return the latest available data buffered at the client side from the most recent update.
ok	Everything is ok and latest data returned.
no data	Client has not yet received an update since subscription and therefore no data is available and no data is returned.
unsubscribed	No data available since client is not subscribed and can thus not receive updates.
disconnected	No data available since client is even not connected to a server.
error	Something went completely wrong and no valid data returned.

Table 5.19: *Get the latest data that is available at the client side.*

getUpdateWait	Blocking call which waits until the next update is received.
ok	Everything is ok and just received data is returned.
cancelled	Blocking not allowed or not allowed anymore. Waiting for the next update is aborted and no valid data is returned.
unsubscribed	Client is unsubscribed or got unsubscribed and member function returns without valid data.
disconnected	Client is not connected or got disconnected and member function returns without valid data.
error	Something went completely wrong and no valid data returned.

Table 5.20: *Wait for the next update at the client side.*

put	Send updated data to all subscribed clients.
ok	Everything is ok.
communication error	Communication problems caused by at least one client. The other clients are updated correctly.
error	Something went completely wrong with at least one client.

Table 5.21: *Provide data at the server side to be sent to all subscribed clients.*

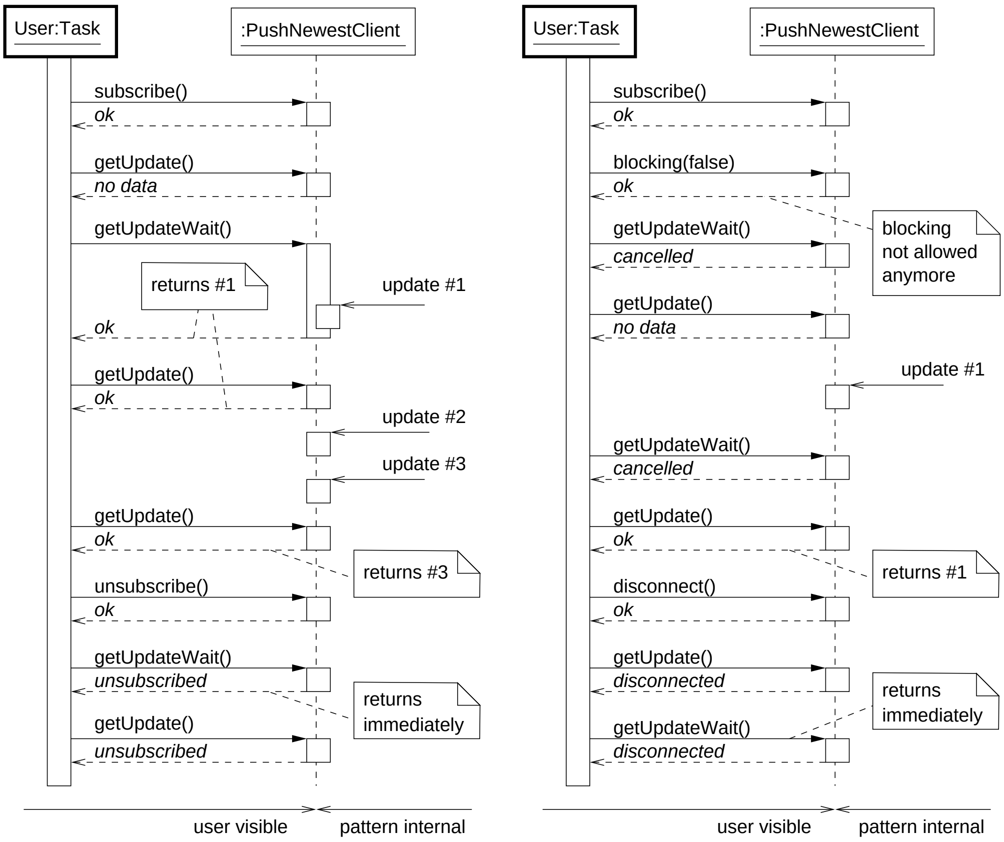


Figure 5.37: *The client side behavior of the user interface of the push newest pattern.*

The Server Side The server side user interface is shown in table 5.21. Calling the *put* member function distributes the updated data to all subscribed clients as shown in figure 5.38.

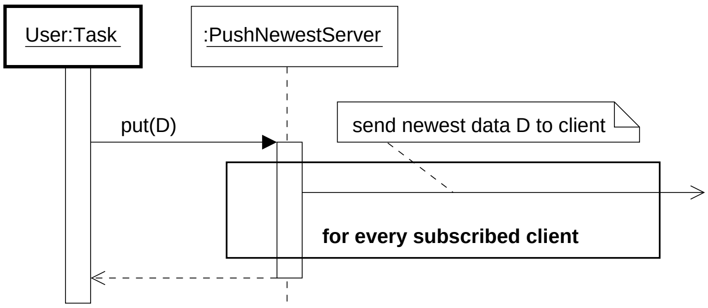


Figure 5.38: *Sequence diagram of the server side operation of the push newest pattern.*

5.5.2.4 The Push Timed Pattern

The basics of the push timed pattern are the same as that of the push newest pattern. The push timed pattern however distributes data on a regular basis and therefore provides an additional mechanism to regularly trigger data distribution. The class diagrams are shown in figure 5.39.

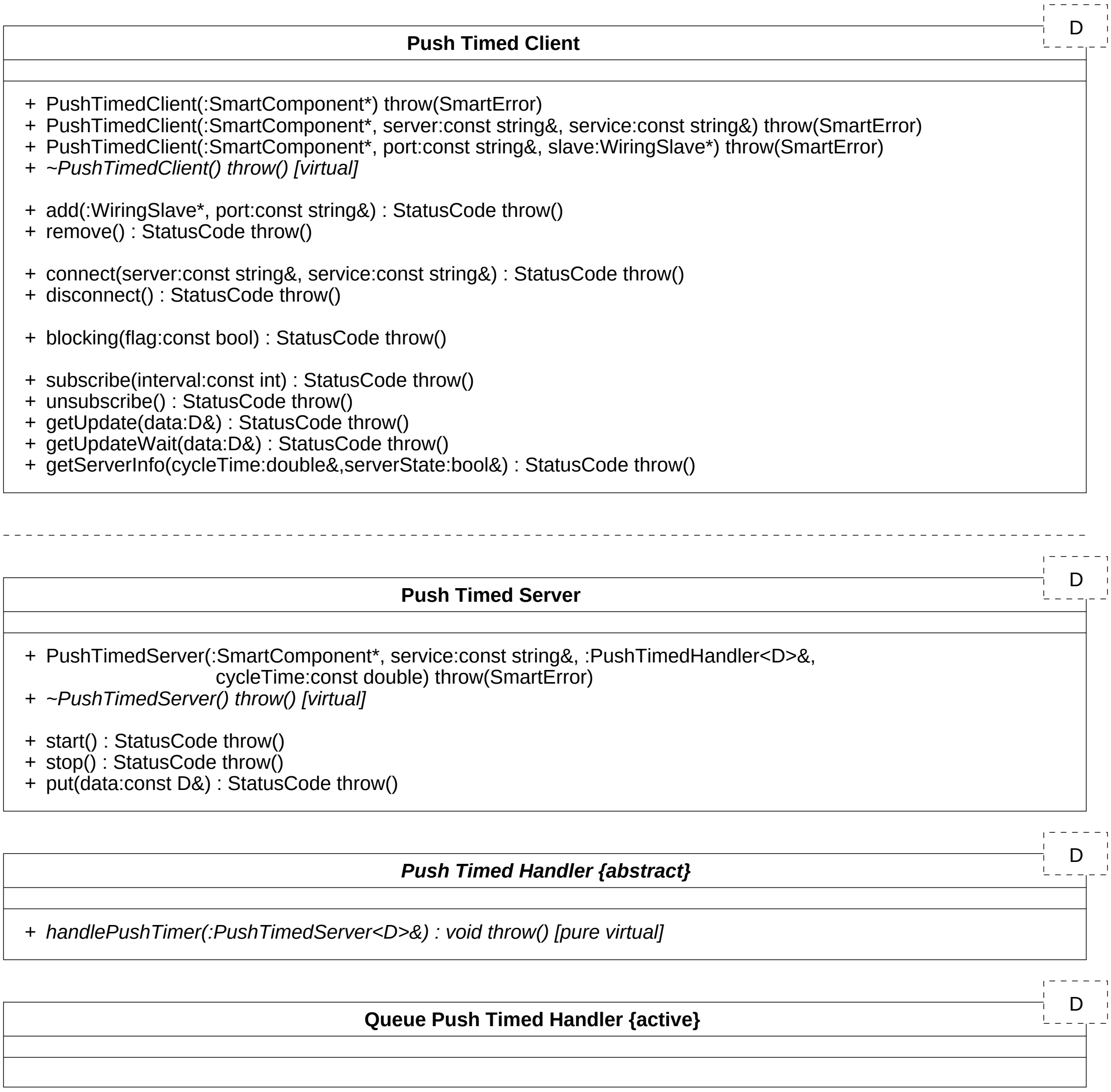

D
Push Timed Client
+ PushTimedClient(:SmartComponent*) throw(SmartError)
+ PushTimedClient(:SmartComponent*, server:const string&, service:const string&) throw(SmartError)
+ PushTimedClient(:SmartComponent*, port:const string&, slave:WiringSlave*) throw(SmartError)
+ ~PushTimedClient() throw() [virtual]
+ add(:WiringSlave*, port:const string&) : StatusCode throw()
+ remove() : StatusCode throw()
+ connect(server:const string&, service:const string&) : StatusCode throw()
+ disconnect() : StatusCode throw()
+ blocking(flag:const bool) : StatusCode throw()
+ subscribe(interval:const int) : StatusCode throw()
+ unsubscribe() : StatusCode throw()
+ getUpdate(data:D&) : StatusCode throw()
+ getUpdateWait(data:D&) : StatusCode throw()
+ getServerInfo(cycleTime:double&,serverState:bool&) : StatusCode throw()
D
Push Timed Server
+ PushTimedServer(:SmartComponent*, service:const string&, :PushTimedHandler<D>&, cycleTime:const double) throw(SmartError)
+ ~PushTimedServer() throw() [virtual]
+ start() : StatusCode throw()
+ stop() : StatusCode throw()
+ put(data:const D&) : StatusCode throw()
D
Push Timed Handler {abstract}
+ handlePushTimer(:PushTimedServer<D>&) : void throw() [pure virtual]
D
Queue Push Timed Handler {active}

Figure 5.39: *The class diagrams of the push timed pattern.*

The Client Side The client can subscribe to get every n-th update which often makes sense if the client otherwise gets overwhelmed by updates. To keep the management of the update rates as simple as possible, only whole-numbered multiples of the server cycle time are supported. The *getServerInfo* member function shown in table 5.26 provides all the necessary information for clients to decide on the appropriate update rate. Both the *getUpdate* and the *getUpdateWait* member functions provide an additional return code indicating that no data is available due to an inactive server. Furthermore, blocking calls are aborted with the *not activated* status code if the server gets deactivated. As can be seen from table 5.25, *getUpdateWait* returns *not activated* instead of *cancelled* for a connected and subscribed client regardless of the blocking mode as soon as the push timed server is deactivated. Again, the status codes are ordered such that only the most causal reason is returned. A disconnect automatically performs an unsubscribe. The client side behavior of the push timed pattern is summarized in figure 5.40.

subscribe	Subscribe at the server to periodically get every n-th update.
ok	Everything is ok and client is subscribed. A newly subscribed client gets the next available new data and is then updated with regard to its individual update cycle.
disconnected	Client is not connected to a server and can therefore not subscribe for updates. Client is still unsubscribed.
communication error	Communication problems and client is not subscribed.
error	Something went completely wrong and client is not subscribed.

Table 5.22: *Subscribe for updates.*

unsubscribe	Unsubscribe to get no more updates.
ok	Everything is ok and client is now unsubscribed or client has already been unsubscribed. All blocking calls are aborted with the appropriate status and yet received and still buffered data is deleted to avoid returning old data.
communication error	Communication problems and client is not unsubscribed.
error	Something went completely wrong and client is not unsubscribed.

Table 5.23: *Unsubscribe to get no more updates.*

getUpdate	Non-blocking call to immediately return the latest available data buffered at the client side from the most recent update.
ok	Everything is ok and latest data returned.
no data	Client has not yet received an update since subscription and therefore no data is available and no data is returned.
not activated	The server is currently not active and does therefore not provide updates at the expected rate. No valid data returned.
unsubscribed	No data available since client is not subscribed and can therefore not receive updates.
disconnected	No data available since client is even not connected to a server.
error	Something went completely wrong and no valid data returned.

Table 5.24: *Get the latest available data.*

The Server Side The server provides a handler which is called each time the next update is due. The handler is operated by the singular component central timer provided by the component management to save resources. Again, a decorator can be used to convert the handler into an active object in case one wants to perform long running activities within the handler. As with the query handler, a separate member function is used to provide the data that is to be distributed to the server pattern as shown

getUpdateWait	Blocking call which waits until the next update is received.
ok	Everything is ok and just received data is returned.
cancelled	Blocking is not allowed or not allowed anymore. Waiting for the next update is aborted and no valid data is returned.
not activated	The server is currently not active and does therefore not provide updates at the expected rate. No valid data returned.
unsubscribed	Returns immediately without data if client is not subscribed.
disconnected	Returns immediately without data since client is even not connected to a server.
error	Something went completely wrong and no valid data returned.

Table 5.25: *Wait for the next update.*

getServerInfo	Get cycle time and server state.
ok	Everything is ok and returned values are valid.
disconnected	Client is not connected to a server and can therefore not get any valid server info.
communication error	Communication problems and returned server info is not valid.
error	Something went completely wrong and no valid server info returned.

Table 5.26: *Get the server state.*

in table 5.29. One of the advantages is that one can use any mechanism to provide the next update without being confined to a handler mechanism. The update cycles are managed by counters that are updated inside the *put* member function. Therefore arbitrarily calling the *put* member function and ignoring the timing as indicated by the time triggered handler would unsettle the update cycles. The *put* member function returns *not activated* in case of a deactivated server. In that case no update is distributed to any clients.

Any state change performed by the *start* and the *stop* member functions is reported to the subscribed clients. If a server gets deactivated, the timer to trigger the handler to invoke the next update is deactivated and blocking calls in subscribed clients are aborted with the *not activated* return code. The client also gets the current server state with every subscription for proper initialization of the client's knowledge on the server state. The activation restarts the timer to trigger the handler and resets all update counters. Therefore all subscribed clients get the first update after activation and thereafter only those fitting into their individual cycles. Figure 5.41 summarizes the server side behavior of the push timed pattern.

start	Activate the server. The activation starts the timer which signals when the next update is due.
ok	Everything is ok.
communication error	Communication problems with at least one client which then is not informed on the server state change.
error	Something went completely wrong with at least one client which then is not informed on the server state change.

Table 5.27: *Activate the server.*

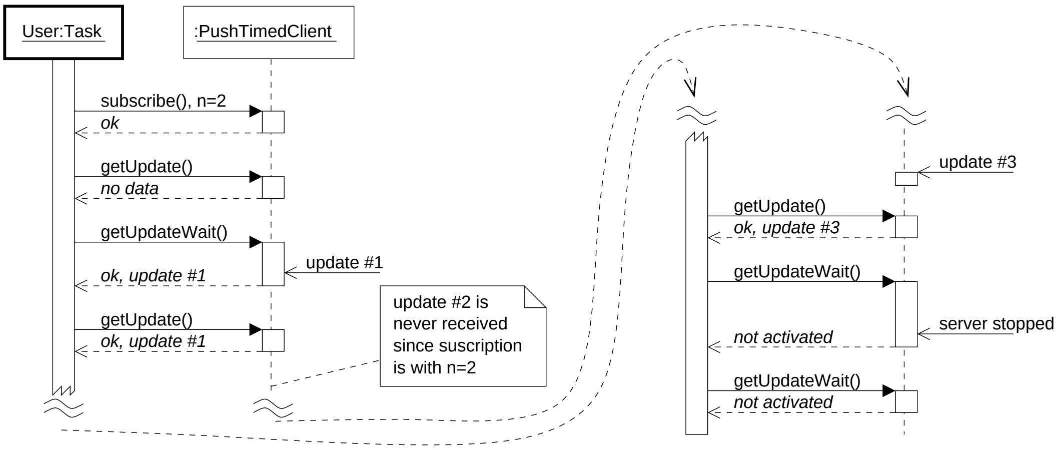


Figure 5.40: *The client side behavior of the user interface of the push timed pattern.*

stop	Deactivate the server. The timer which signals when the next update is due is stopped.
ok	(see above)
communication error	(see above)
error	(see above)

Table 5.28: *Deactivate the server.*

put	Provide new data which is sent to all subscribed clients taking into account their individual update rates.
ok	Everything is ok.
not activated	Server is stopped and does therefore not distribute any data to clients. In that case, update interval counters are not touched.
communication error	Communication problems caused by at least one client. The other clients are updated correctly.
error	Something went completely wrong with at least one client.

Table 5.29: *Provide data for subscribed clients taking into account their individual update rates.*

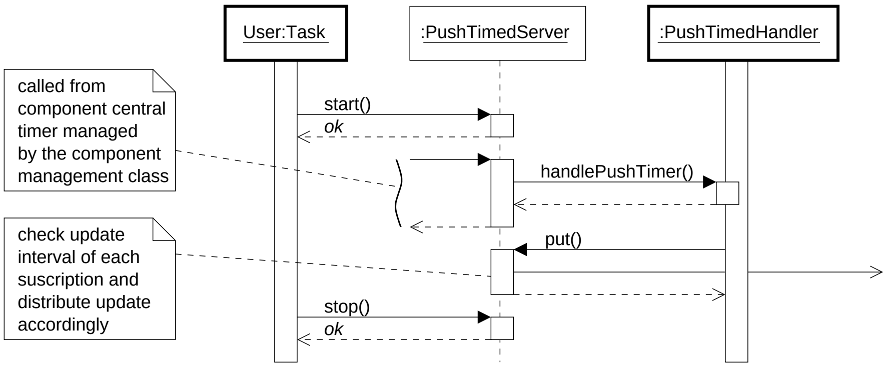

User:Task
:PushTimedServer
:PushTimedHandler
called from component central timer managed by the component management class
start()
ok
handlePushTimer()
check update interval of each suscription and distribute update accordingly
put()
stop()
ok

Figure 5.41: *The server side behavior of the user interface of the push timed pattern with an active handler.*

5.5.2.5 The Event Pattern

The user interface of the event pattern is summarized in the class diagrams in figure 5.42. The client part is parameterized by the communication objects for the activation parameters *P* and for the firing

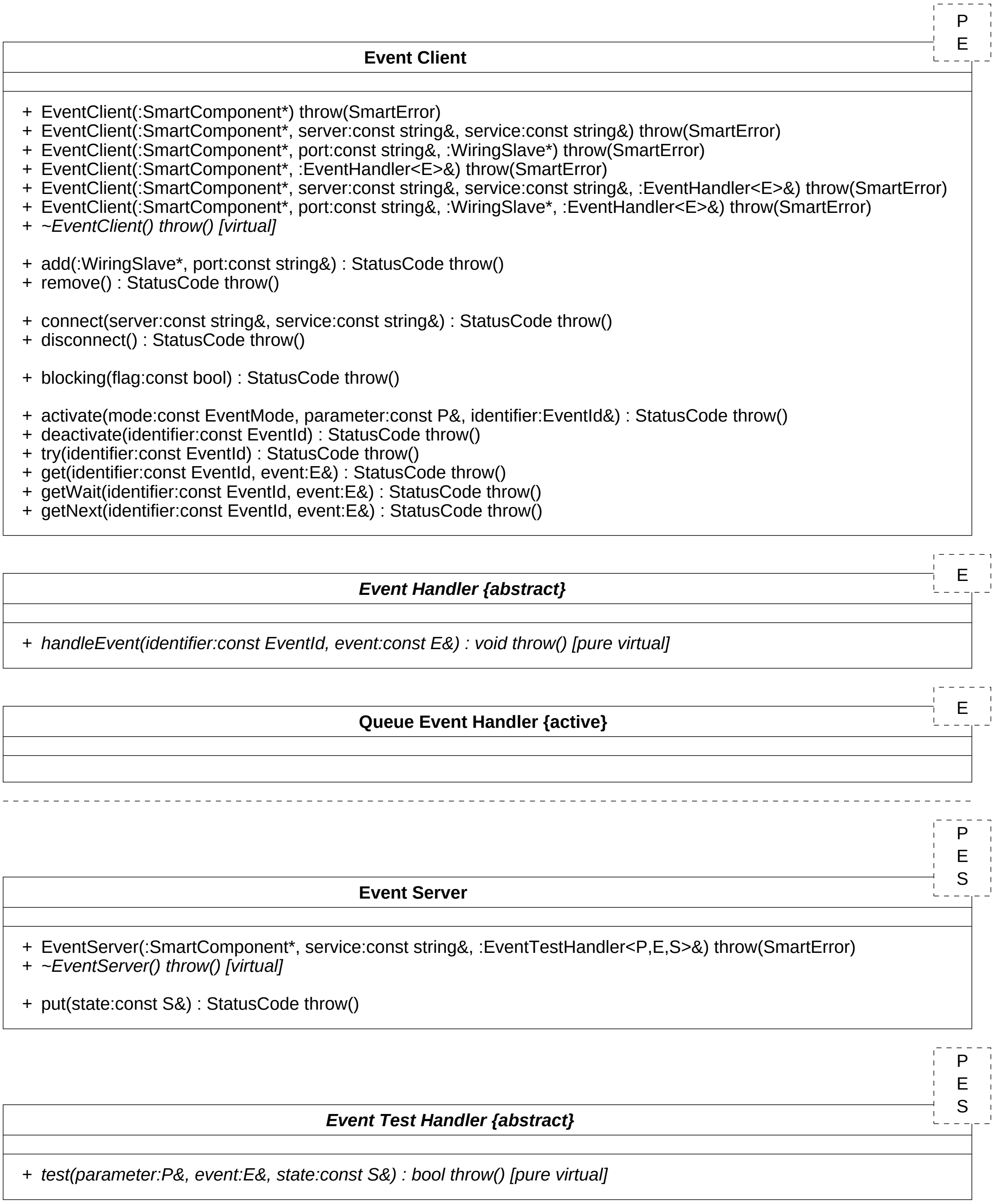

P
E
Event Client
+ EventClient(:SmartComponent*) throw(SmartError)
+ EventClient(:SmartComponent*, server:const string&, service:const string&) throw(SmartError)
+ EventClient(:SmartComponent*, port:const string&, :WiringSlave*) throw(SmartError)
+ EventClient(:SmartComponent*, :EventHandler<E>&) throw(SmartError)
+ EventClient(:SmartComponent*, server:const string&, service:const string&, :EventHandler<E>&) throw(SmartError)
+ EventClient(:SmartComponent*, port:const string&, :WiringSlave*, :EventHandler<E>&) throw(SmartError)
+ ~EventClient() throw() [virtual]
+ add(:WiringSlave*, port:const string&) : StatusCode throw()
+ remove() : StatusCode throw()
+ connect(server:const string&, service:const string&) : StatusCode throw()
+ disconnect() : StatusCode throw()
+ blocking(flag:const bool) : StatusCode throw()
+ activate(mode:const EventMode, parameter:const P&, identifier:EventId&) : StatusCode throw()
+ deactivate(identifier:const EventId) : StatusCode throw()
+ try(identifier:const EventId) : StatusCode throw()
+ get(identifier:const EventId, event:E&) : StatusCode throw()
+ getWait(identifier:const EventId, event:E&) : StatusCode throw()
+ getNext(identifier:const EventId, event:E&) : StatusCode throw()
E
Event Handler {abstract}
+ handleEvent(identifier:const EventId, event:const E&) : void throw() [pure virtual]
E
Queue Event Handler {active}
P
E
S
Event Server
+ EventServer(:SmartComponent*, service:const string&, :EventTestHandler<P,E,S>&) throw(SmartError)
+ ~EventServer() throw() [virtual]
+ put(state:const S&) : StatusCode throw()
P
E
S
Event Test Handler {abstract}
+ test(parameter:P&, event:E&, state:const S&) : bool throw() [pure virtual]

Figure 5.42: *The class diagrams of the event pattern.*

activation E. The server part additionally requires the state description S which is the basis for testing the event predicate. The event predicate is checked at the server each time a new state description is provided. An event can have multiple activations even from different clients each with their own parameters for the event predicate. Each activation is checked individually and fires only according to the individual activation parameters. An individual event can therefore be used to report on different events. For example, an event which can check a value against a threshold can be activated multiple times with customized thresholds to report on different levels reached.

	stateless	**with state**
single	Fire once if a specific state is true.	Fire once as soon as a specific state change is performed.
use case	Fire when the robot is inside a specific region. The event also fires with the next test if the robot is already inside the region.	Fire when the robot performs a specific state change, for example, by entering a specific region. The event fires only if the correct state change occurred between two tests.
continuous	Fire as long as a specific state is true.	Fire with every state change.
use case	Fire as long as the robot is inside a specific region and report the current position of the robot with every firing.	Fire each time a state change occurs, for example, by entering or leaving a specific region but don't fire if nothing changes.

Table 5.30: *The various event modi with use cases.*

Each activation can be individually set to either *single* or *continuous* mode. In *single* mode, it is the event pattern that makes sure that an activation fires only once irrespective of the results of the event predicate at further tests. The event predicate additionally allows modifications of the activation parameters which is useful to implement a state based event by memorizing states in the parameter object. Table 5.30 summarizes the resulting event behaviors.

The Client Side The event activation is shown in table 5.31. It expects the event mode and the parameters for the server side event predicate and returns a unique activation identifier. The event mode is defined separately for every activation. The identifier is further used for referencing, stays valid till deactivation and should not be used in concurrent blocking method calls. Since a blocking method consumes an event, only one call returns the firing activation and the others return *lost*.

activate	Activate an event with the provided parameters in either *single* or *continuous* mode.
ok	Event is activated and a valid identifier is returned.
disconnected	Activation not possible since not connected to a server. No valid identifier returned.
communication error	Communication problems, event not activated and no valid identifier returned.
error	Something went completely wrong, event not activated and no valid identifier returned.

Table 5.31: *Event activation with individual event parameters and individual event mode.*

Deactivation of an event is summarized in table 5.32. An event must always be deactivated explicitly and is only deactivated automatically with a disconnect. A disconnect deactivates all events since activations are related to the service provider at which the activation was performed and therefore get invalid with a disconnect. The client provides both a member function based interface and a handler based interface.

deactivate	Deactivate the event with the specified identifier.
ok	Everything is ok and event is deactivated. The identifier is not valid anymore.
wrong identifier	There is no activation available with this identifier (still connected to service provider).
communication error	Communication problems, deactivation not properly completed.
error	Something went completely wrong, deactivation not properly completed.

Table 5.32: *Deactivation of an event.*

try	Solely check the state of the event.
single mode	
ok	Event fired already, event is still available and can be consumed by calling *get* or *getWait*.
active	Event has not yet fired.
passive	Event fired already and event was already consumed.
wrong identifier	No activation available with this identifier.
continuous mode	
ok	Unconsumed event is available. Since events are overwritten this means that at least one new event has been received since the last event consumption.
active	Currently there is no unconsumed event available.
wrong identifier	(see above)

Table 5.33: *Check whether an event activation fired already.*

get	Non-blocking call to consume an event.
single mode	
ok	Event fired already but event has not yet been consumed. Thus, event is consumed and returned. Only the *ok* status indicates that a valid event is returned.
active	Event has not yet fired.
passive	Event fired and event got consumed already.
wrong identifier	No activation available with this identifier.
continuous mode	
ok	Unconsumed event is available and event is consumed and returned. Due to the overwriting behavior, only the latest firing of the event is available.
active	There is no unconsumed event available.
wrong identifier	(see above)

Table 5.34: *Non-blocking call to consume an event.*

getWait	Return immediately in case of an unconsumed event and otherwise wait till the denoted activation fires again.
single mode	Since an event in single mode fires only once, return immediately if the event got consumed already. Also return immediately if the event fired already but the event has not yet been consumed. Otherwise wait till the event fires.
ok	Unconsumed event is either already available or got available while waiting. Event is consumed and returned. Only the *ok* status indicates that a valid event is returned.
wrong identifier	No activation available with this identifier.
passive	Event fired and got consumed already. Return immediately since event cannot fire again in single mode.
cancelled	Event has not yet fired and blocking is already not allowed at the time of method invocation or waiting for the event to fire has been aborted since blocking is not allowed anymore.
lost	The event fired while waiting but a concurrent call with the same activation identifier consumed the event.
not activated	Got deactivated while waiting. Thus, identifier is not valid any longer.
disconnected	Client got disconnected while waiting. Thus, identifier is not valid any longer.
continuous mode	Returns immediately if an unconsumed event is available. Otherwise, wait till the event fires again. Due to the overwriting behavior, only the latest firing of the event can be available as unconsumed event.
ok	(see above)
wrong identifier	(see above)
cancelled	(see above)
lost	(see above)
not activated	(see above)
disconnected	(see above)

Table 5.35: *Wait for an event activation to fire respectively consume already fired events.*

getNext	Blocking call which waits for the *next* arriving event to make sure that only events arriving after entering this member function are considered. Event is consumed and events received before calling this member function are ignored.
single mode	In single mode one misses the event if it fired before entering this member function.
ok	Event fired while waiting and event is consumed and returned. Only the *ok* status indicates that a valid event is returned.
wrong identifier	(see above)
passive	Event fired prior to calling this member function. Independently of whether the event is already consumed or not, there can be no next firing. Thus, return immediately without a valid event. Of course, an unconsumed event can still be consumed by calling *get* or *getWait*.
cancelled	(see above)
lost	(see above)
not activated	(see above)
disconnected	(see above)
continuous mode	Makes sure that only firings after entering this member function are considered.
ok	Event fired while waiting and event is consumed and returned. Only the *ok* status indicates that a valid event is returned.
wrong identifier	(see above)
cancelled	Blocking is already not allowed at the time of method invocation or waiting for the event to fire the next time has been aborted since blocking is not allowed anymore.
lost	The event fired while waiting but a concurrent call with the same activation identifier consumed the event.
not activated	(see above)
disconnected	(see above)

Table 5.36: *Wait until an event activation fires after invoking this member function.*

The member function based interface of the client part is summarized in tables 5.33, 5.34, 5.35 and 5.36. These member functions allow to check whether an event already fired and provide blocking and non-blocking member functions to get an unconsumed event or to wait till the event fires or till it fires again. Their behavior differs with respect to the event mode. For example, waiting for an event to fire again makes no sense in *single* mode and thus, the corresponding methods do not block.

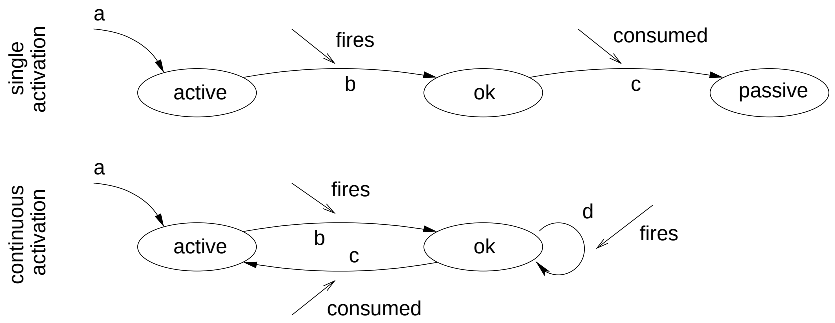


Figure 5.43: *The client side behavior of an event activation.*

Figure 5.43 illustrates the overall behavior of the client side. In *single* mode, the state of the event activation is *active* if the activation has not yet fired and it is *ok* if the activation has already fired but has not yet been consumed. Consuming a fired activation performs a state change from *ok* to *passive*. The *try* member function only checks the state of the activation and always returns immediately with the appropriate status code and never consumes a fired activation. The *get* member function behaves like the *try* member function but consumes a fired activation in case there is an unconsumed one available. Both, the *getWait* and the *getNext* member function only block when called in the *active* state. The *getWait* member function returns the unconsumed event when called in the *ok* state whereas the *getNext* member function indicates that the firing of the activation has been missed. Concurrent calls to *getWait* and *getNext* with the same activation identifier are all released with a firing activation, but only one call consumes the fired activation and returns *ok* and the others return appropriate status codes.

The *continuous* mode works similar. The *active* state indicates that there is no unconsumed event available and *ok* that there is one available. An unconsumed event can be consumed either by the *get* or the *getWait* method. The latter does not block in case it is called in the *ok* state whereas *getNext* always waits for the next firing. Consuming a fired activation switches from the *ok* to the *active* state and calls to *getWait* and *getNext* block again. An activation that fires multiple times in the *ok* state only holds the latest firing and event firings not yet consumed get overwritten, Again, any kinds of concurrent calls to *get*, *getWait* and *getNext* with the same activation identifier are sorted out without leaving a blocking call behind but only one invocation consumes the event.

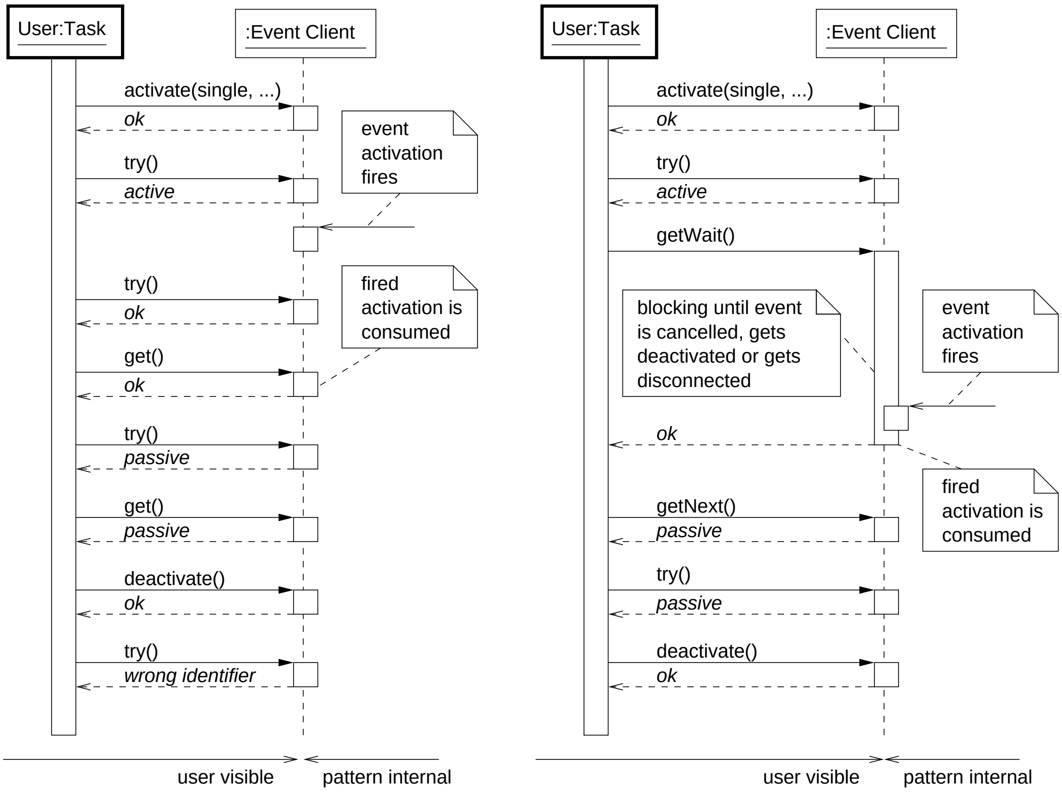


Figure 5.44: *Sequence diagram of user interface of client side with* single *mode activation.*

Storing only the latest firing of an activation in *continuous* mode avoids boundless growing buffers. The users have to either use the handler based interface if no firing can be missed in *continuous* mode or to simply specify the returned event objects in such a way that they contain states instead of state transitions. Then the latest event contains all necessary information irrespective of previous firings. Figures 5.44 and 5.45 show sequence diagrams to further detail the client side behavior.

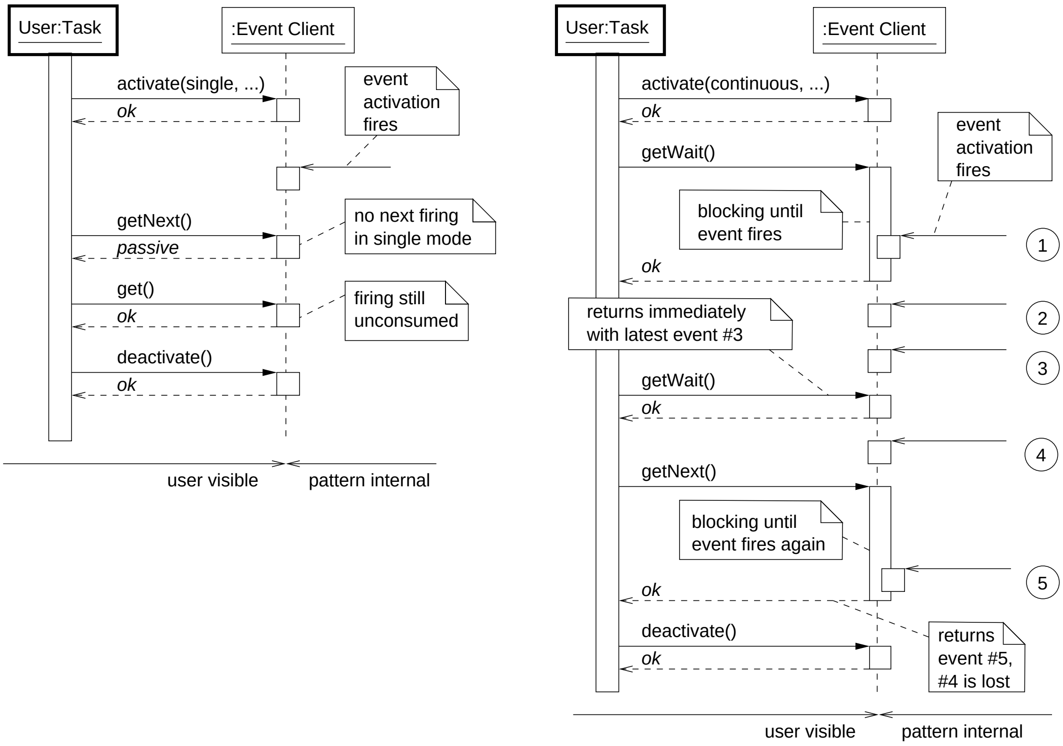


Figure 5.45: *Sequence diagram of user interface of client side with* single *and with* continuous *mode activation.*

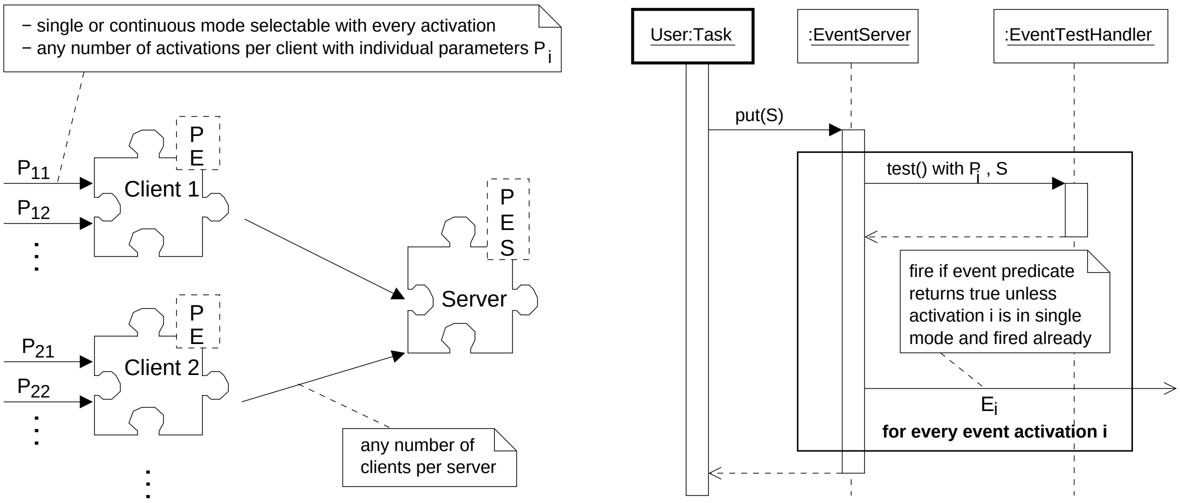
The handler based interface is particularly useful if one wants to implement a central dispatcher or if the pattern is used to asynchronously report events and one does not want to wait for the next firing of the event. Once a handler is provided at the client side, every incoming event is forwarded to the handler. Individual activations can always be distinguished by the unique activation identifier which is also forwarded to the handler. The *Queue Event Handler* is a decorator for the client side handler to convert the handler into an active object where the *handleEvent* member function is executed by a separate thread.

The Server Side The server part requires a handler derived from the abstract *event test handler* to implement the event predicate in its *test* member function. The event predicate checks the event condition based on the activation parameters *P* and the current state *S*. The state *S* is provided by the user by calling the *put* member function shown in table 5.37. Returning *true* from the *test* member function causes the tested activation to fire if that is compatible to the event mode. The event object *E* can be set inside the event predicate and normally grabs a snapshot of all relevant information to exactly describe the circumstances which caused the event to fire. In case the event fires, the event object *E* is transmitted to the appropriate client. The *test* member function can furthermore change the activation parameters *P* to save information perhaps needed with the next predicate evaluation. For example, one can implement a state automaton with the state stored in the activation parameters as illustrated in the example in the next paragraph.

The operating sequence of the server side of the event pattern is summarized in figure 5.46. The server calls the event predicate separately for every activation from within the *put* member function as soon as a to be checked state description *S* is provided.

put	Initiate testing the event conditions for the activations.
ok	Everything is ok and all event activations successfully checked with the new state.
communication error	Communication problems with at least one of the clients that should have been notified of a firing event activation.
error	Something went completely wrong, see *communication error*.

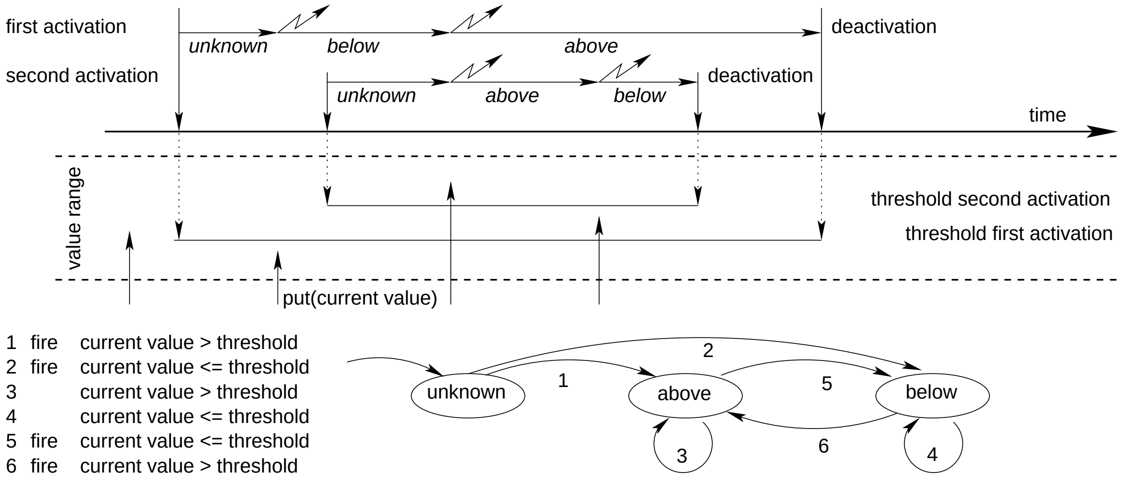
Table 5.37: *Providing a new and to be tested state to the event server.*



− single or continuous mode selectable with every activation
− any number of activations per client with individual parameters P i
P11
P12
P E
Client 1
P21
P22
P E
Client 2
P E S
Server
any number of clients per server
User:Task
:EventServer
:EventTestHandler
put(S)
test() with Pi , S
fire if event predicate returns true unless activation i is in single mode and fired already
Ei
for every event activation i

Figure 5.46: *The operating sequence of the server side of the event pattern. Each activation provides its individual parameters P_i which are then checked at the server side with every new state S.*

Example Figure 5.47 shows an example of the implementation of a state based event with two continuous activations. The activation parameters provide an individual threshold and a state flag that is initialized to *unknown*. This makes sure that the first call to the event predicate always reports the current state to the client. The event predicate operates the state automaton with the state stored in the activation parameters and returns *true* to fire the event only with the state transitions 1, 2, 5 and 6. The state stored in the activation parameters allows individual states for every activation with an automaton provided by the event predicate.



1	fire	current value > threshold
2	fire	current value <= threshold
3		current value > threshold
4		current value <= threshold
5	fire	current value <= threshold
6	fire	current value > threshold

Figure 5.47: *Example of handling continuous activations of an event with state at server side.*

5.5.2.6 The Wiring Pattern

The wiring pattern supports dynamic wiring of services from outside a component by exposing service requestors as ports. The user interface of the wiring pattern is summarized in the class diagrams in figure 5.48.

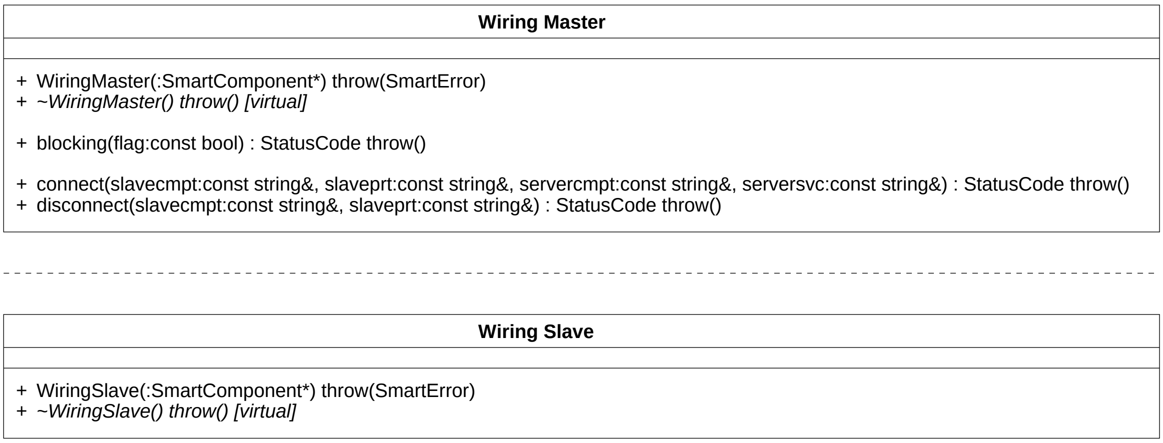

Wiring Master
+ WiringMaster(:SmartComponent*) throw(SmartError)
+ ~WiringMaster() throw() [virtual]
+ blocking(flag:const bool) : StatusCode throw()
+ connect(slavecmpt:const string&, slaveprt:const string&, servercmpt:const string&, serversvc:const string&) : StatusCode throw()
+ disconnect(slavecmpt:const string&, slaveprt:const string&) : StatusCode throw()
Wiring Slave
+ WiringSlave(:SmartComponent*) throw(SmartError)
+ ~WiringSlave() throw() [virtual]

Figure 5.48: *The class diagrams of the wiring pattern.*

The wiring pattern is different to the other communication patterns in not requiring communication objects. Furthermore, the wiring slave does not possess user callable member functions and operates transparently for the user without requiring user interactions. The *connect/disconnect* member functions of the wiring master shown in tables 5.38 and 5.39 must not be confused with the conforming member functions of the service requestors of the other patterns. The *connect* member function does not establish a connection from the wiring master to the wiring slave but connects a port with a service provider. A wiring master can wire arbitrary ports of arbitrary components without being explicitly connected to a wiring slave before issuing a wiring command. The status codes of the *connect/disconnect* member functions distinguish three different levels. The *unknown component* status code is related to connecting to the wiring slave and is returned if the wiring master is unable to connect itself to the slave component because that component either is unknown or does not possess a wiring slave. *Unknown port* is related to the slave component and is returned if the wiring master was able to connect to the slave component but no such port is known there. The other status codes are related to the service provider to which the port is to be connected to.

The integration of the wiring pattern into a component is illustrated in figure 5.49. Applying the wiring pattern only requires to instantiate the wiring classes. A component can have at most one wiring slave. Service requestors can themselves register at the component central wiring slave using their *add/remove* member functions. Independently of being exposed as port, one can still use the *connect/disconnect* member functions of the service requestors to change wirings. The finally effective wiring is solely determined by the order of the calls. A wiring master can of course also be used to configure the ports of its component.

5.5.3 The Communication Objects

Communication objects represent the commonly agreed data structures that are exchanged between components via the communication patterns. Component builders normally reuse communication

connect	Blocking member function to connect port *slaveport* of component *slavecmpt* with service *serversvc* of component *servercmpt*.
ok	Everything is ok and the requested connection has been established successfully. An old connection is first removed before a new connection is established.
cancelled	Blocking is not allowed or is not allowed anymore and therefore blocking call is aborted. The port of the slave component can now be either unchanged, disconnected or properly connected to the specified service provider.
unknown component	The addressed slave component is either not known or does not provide a wiring service. The requested connection cannot be established.
unknown port	The specified port name is not known at the slave component. The requested connection cannot be etablished.
service unavailable	The slave component cannot connect to the specified service of the specified server since the server and/or the service to be connected to from the slave component is not available.
service incompatible	The service behind the specified port is not compatible with the service to be connected to. Requested connection cannot be established.
disconnected	The addressed wiring slave got destroyed while wiring was in progress.
communication error	Communication problems either while connecting to the slave or at the slave component while it tried to establish the requested connection to the requested service provider. The port of the slave component now can either be unchanged, disconnected or already properly connected to the specified service provider.
error	Something went completely wrong. See *communication error*.

Table 5.38: *Connect a port to a service provider.*

disconnect	Blocking member function to disconnect port *slaveport* of component *slavecmpt* from service provider.
ok	Everything is ok and the port is disconnected.
cancelled	Blocking is not allowed or is not allowed anymore and therefore blocking call is aborted. The port of the slave component now can either be unchanged or disconnected.
unknown component	(see above), requested disconnect not performed.
unknown port	(see above), requested disconnect not performed.
disconnected	(see above), port now can either be unchanged or disconnected.
communication error	(see above), port now can either be unchanged or disconnected.
error	(see above), port now can either be unchanged or disconnected.

Table 5.39: *Disconnect a port from its service provider.*

objects as often as possible to minimize the number of different representations. Therefore, even component builders normally only add a small number of new objects to the pool of already available communication objects. Reusing communication objects reduces the effort of applying communication patterns to binding and instantiating templates.

It is important to recognize that communication objects are regular objects decorated with only three additional member functions which implement the framework interface. Adding the interface member functions extends any object to a communication object which then can be used with the communication patterns. The additional interface consists of a *name*, a *get* and a *set* member function. The *name* member function returns a name to identify the communication object type. It has to be unique within the namespace of communication objects. The *get* and *set* member functions implement the marshalling of the content to be transmitted. Marshalling is the process of gathering data from eventually non-contiguous sources in computer storage and converting the data into a platform independent and contiguous representation that can be transmitted across computer architectures and

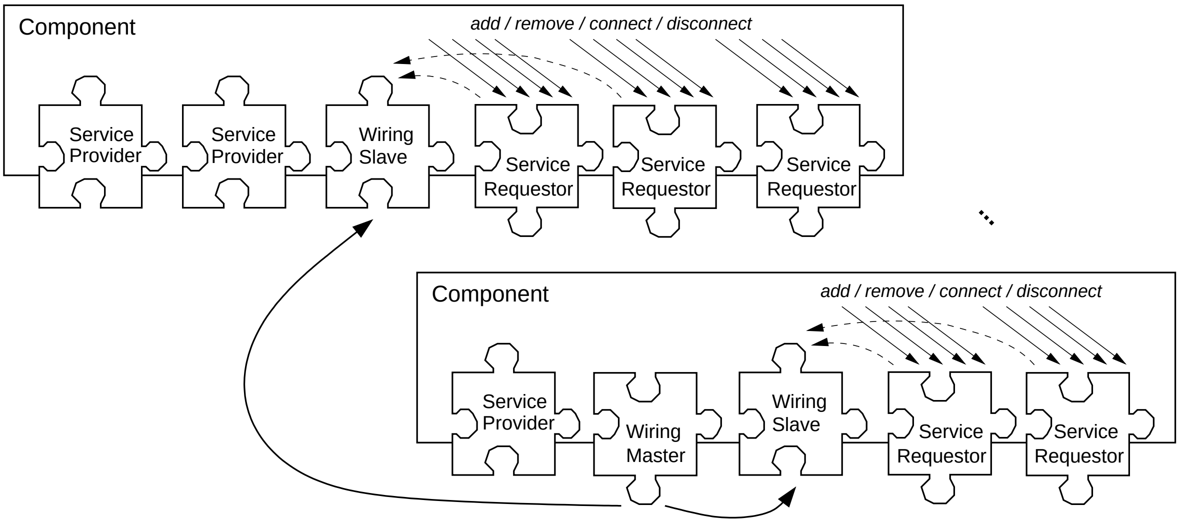


Figure 5.49: *Applying the wiring pattern in components.*

operating systems.

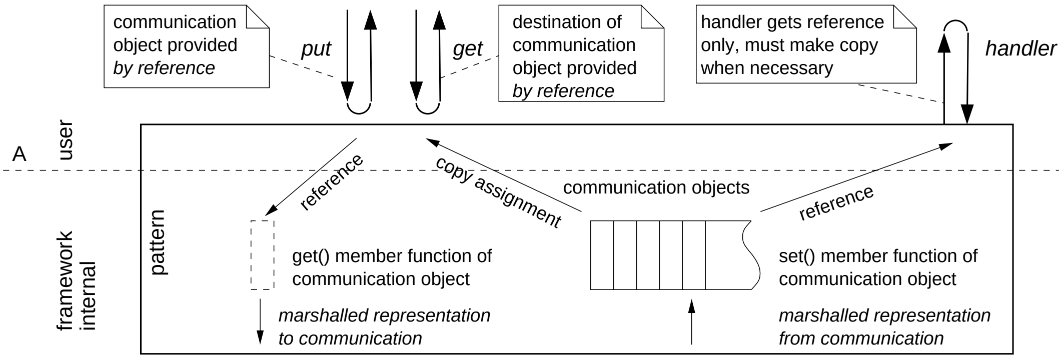

communication object provided *by reference*
put
get
destination of communication object provided *by reference*
handler gets reference only, must make copy when necessary
handler
A
user
framework internal
pattern
reference
copy assignment
communication objects
reference
get() member function of communication object
set() member function of communication object
marshalled representation to communication
marshalled representation from communication

Figure 5.50: *Communication objects and marshalling. The dotted stripline separates the framework internal level from the user visible part and is labeled with A in all figures from now on.*

Figure 5.50 illustrates the principle interaction of a communication pattern with a communication object. Communication patterns always hold their own communication object instances, either by copying or by a life cycle management for references based on reference counters. The underlying communication layer never gets in touch with communication objects besides the interface member functions of the communication objects provided as framework interface. The communication layer therefore only sees marshalled representations.

Marshalling can be implemented in many different ways and is needed anyway and independently of the underlying communication mechanism. All full-fledged communication systems provide marshalling mechanisms which however are almost never interoperable. Further abstraction of the marshalling mechanism is by purpose not considered since wrapping then would introduce additional overhead on top of an already resource intensive process. Instead, the marshalling mechanism inside the communication objects is exposed to the component builder via the *get* and the *set* member functions of the communication objects. Those member functions are however the only place where one gets in touch with aspects of the underlying communication and they are only relevant to component builders who introduce new communication object types.

Asking the component builder to implement the *get* and the *set* member functions instead of generating them automatically also has significant advantages. One is completely free to use any mechanism which provides a representation which can be transmitted transparently across different operating systems and host architectures. For example, the *get* and *set* member functions can also deal with data structures stored in dynamic memory. The *get* method extracts the relevant content and the *set* method rebuilds the structure in dynamic memory. That is possible because the semantics of the communication patterns always is to transfer communication objects *by value*. Therefore, pointers inside communication objects are only a matter of memory organization and are not expected to stay valid across component boundaries. Furthermore, one can also transmit selected data structures and rebuild alternative representations locally instead of always transmitting the whole content of a communication object. For example, with a laser range scan, one can solely transmit the polar scan and a pose to indicate where that scan was taken instead of including the cartesian scan. It is now up to the communication object to internally convert the polar scan into a cartesian scan upon request.

The state-of-the-art implementation of the proposed approach uses *CORBA* as communication mechanism and therefore naturally uses the *CORBA IDL* [28] to describe the format of the data structures that are to be transmitted. It is important to notice that the *IDL* is only used to benefit from the automatically generated marshalling operators. The *CORBA IDL* compiler overloads the `<<=` and `>>=` operators for type safe conversions between the *IDL* described data type used inside a communication object and a `CORBA::Any` type transmitted via the communication layer. The implementation of the *get* and *set* member functions of the *CORBA* based version simply consists of applying the `<<=` and `>>=` operators to perform marshalling.

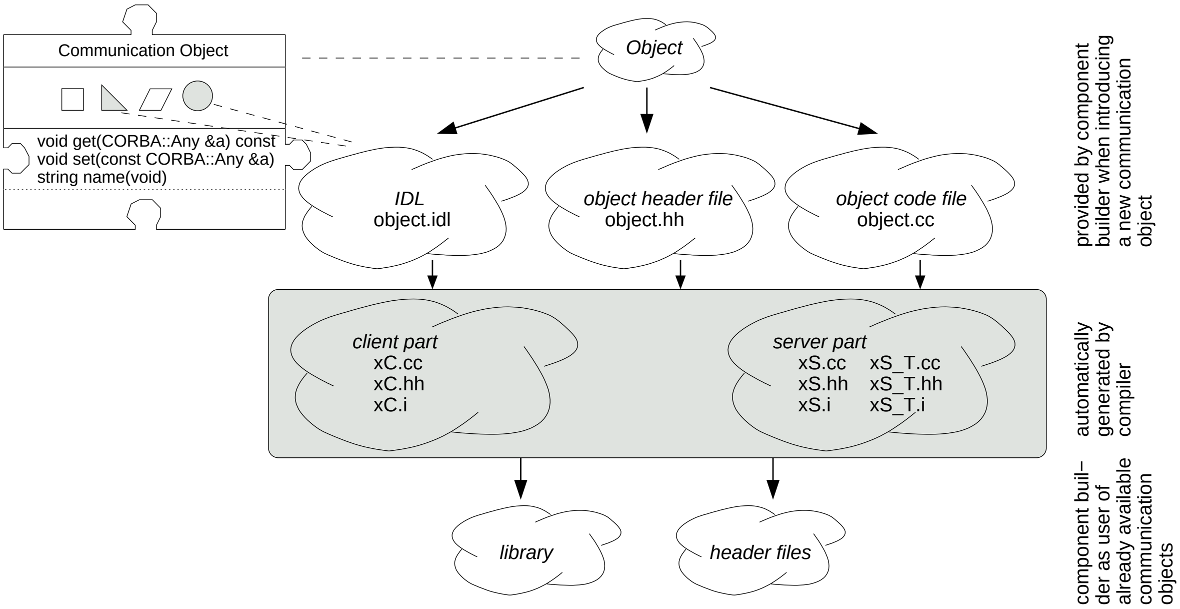

Communication Object
void get(CORBA::Any &a) const
void set(const CORBA::Any &a)
string name(void)
Object
IDL
object.idl
object header file
object.hh
object code file
object.cc
client part
xC.cc
xC.hh
xC.i
server part
xS.cc xS_T.cc
xS.hh xS_T.hh
xS.i xS_T.i
library
header files
provided by component builder when introducing a new communication object
automatically generated by compiler
component buil– der as user of already available communication objects

Figure 5.51: *Marshalling with the CORBA based implementation. The IDL does not describe the object but only the data structures to be transmitted.*

The *CORBA IDL* is not used to describe the overall communication object since this would restrict the data types of the member functions of the communication objects to the *CORBA* types. These member functions and their data types are completely independent of the modelling capabilities of the underlying communication mechanism. Communication objects are the place where the adjustment of communicated data structures to those used by the user takes place. This of course does not mean that one cannot completely describe a communication object by the *CORBA IDL* if one accepts the restrictions then imposed on the data types of the member functions.

The process of implementing a communication object using the *CORBA IDL* is illustrated in figure 5.51. A regular object is decorated by the framework interface and data structures relevant for transmission are described by the *IDL*. The communication object is then compiled into a library which has to be linked to a component if that communication object is being used.

Implementing the proposed approach on top of another communication mechanism instead of using *CORBA* might require more effort when implementing the *get* and *set* member functions. In principle, however, there is no difference and one could even use libraries which provide *CORBA* compatible marshalling functions without having to use an *IDL* compiler.

5.5.4 Example of Usage

The following example illustrates the usage of the communication patterns and illustrates how easily they can be applied. The example uses the *CORBA* based implementation. As component builder, one normally first checks the availability of standardized and reusable communication objects before defining a component interface. Figure 5.52 shows parts of the hierarchy of transmittable data structures used inside standardized communication objects for robotic applications. Hierarchically composing those data structures avoids introducing varying representations of one and the same content. Figure 5.53 shows the implementation of a simple communication object containing a time stamp. The member functions implement the mandatory framework interface and an arbitrary user interface. Figure 5.54 shows the basic structure of the more complex communication object for laser scans taken by a laser range finder mounted horizontally on a moving platform.

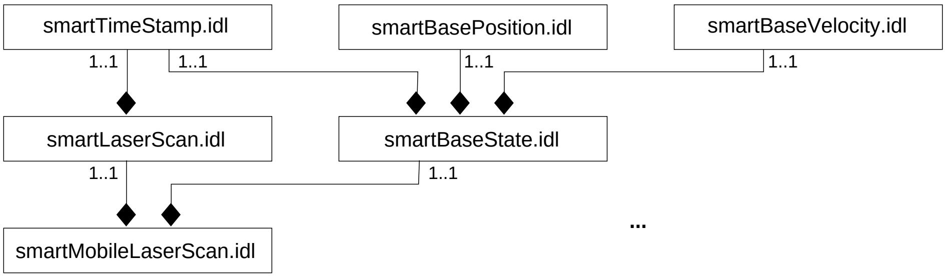
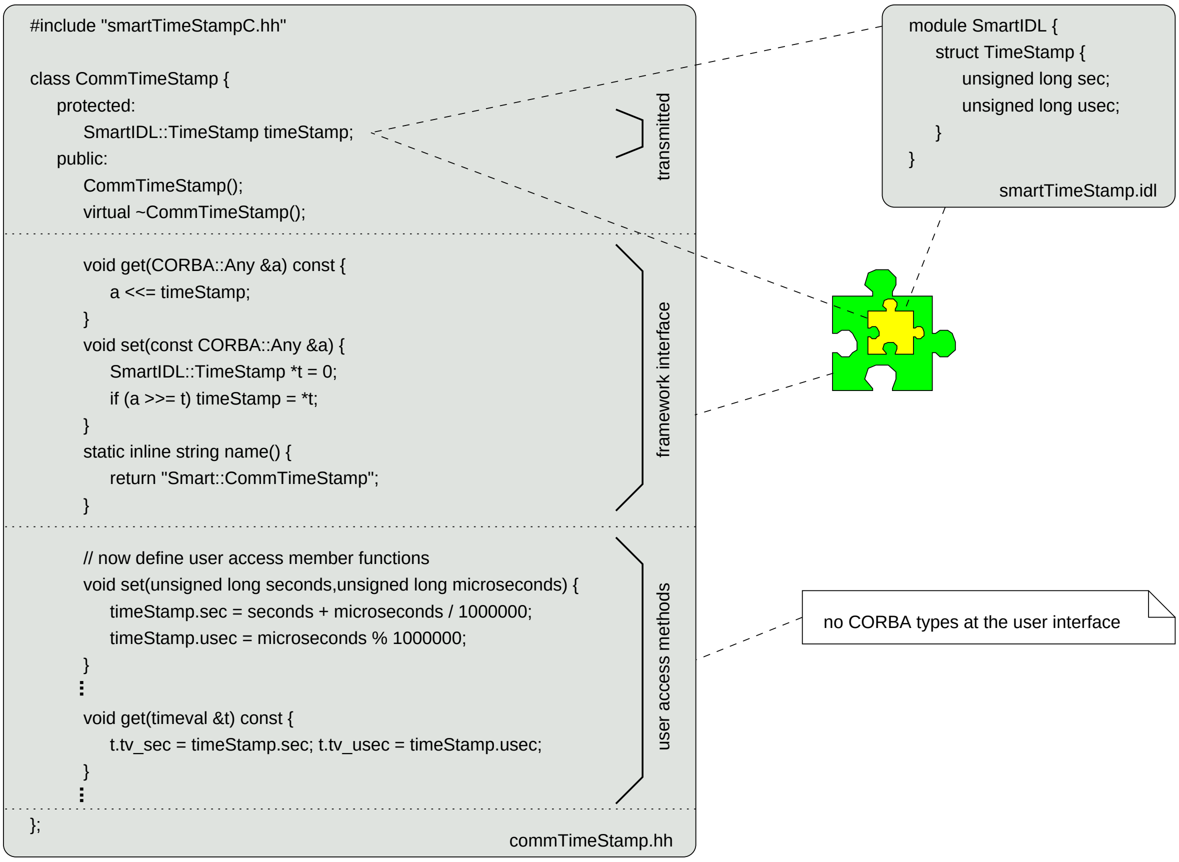


Figure 5.52: *Class diagram of parts of the basic data structures used inside communication objects for robotics.*

The following two example components show the usage of the *query* pattern. The components are kept as simple as possible and require respective provide only one service besides the wiring service. The component shown on the left side of figure 5.55 is named *first* and contains the client part of the *query* pattern. The *query* pattern is bound by a *void* object for invoking a request without further parameters and by a communication object for laser scans containing the answer. The service requestor is visible as port named *laserPort*. The user activities are performed by a separate thread. The main thread operates the *run* member function of the component management to keep the SMARTSOFT framework alive. The service requestor of component *first* can be wired from outside using



```
#include "smartTimeStampC.hh"

class CommTimeStamp {
    protected:
        SmartIDL::TimeStamp timeStamp;
    public:
        CommTimeStamp();
        virtual ~CommTimeStamp();

        void get(CORBA::Any &a) const {
            a <<= timeStamp;
        }
        void set(const CORBA::Any &a) {
            SmartIDL::TimeStamp *t = 0;
            if (a >>= t) timeStamp = *t;
        }
        static inline string name() {
            return "Smart::CommTimeStamp";
        }

        // now define user access member functions
        void set(unsigned long seconds,unsigned long microseconds) {
            timeStamp.sec = seconds + microseconds / 1000000;
            timeStamp.usec = microseconds % 1000000;
        }
        ⋮
        void get(timeval &t) const {
            t.tv_sec = timeStamp.sec; t.tv_usec = timeStamp.usec;
        }
        ⋮
};
```

```
module SmartIDL {
    struct TimeStamp {
        unsigned long sec;
        unsigned long usec;
    }
}
```

no CORBA types at the user interface

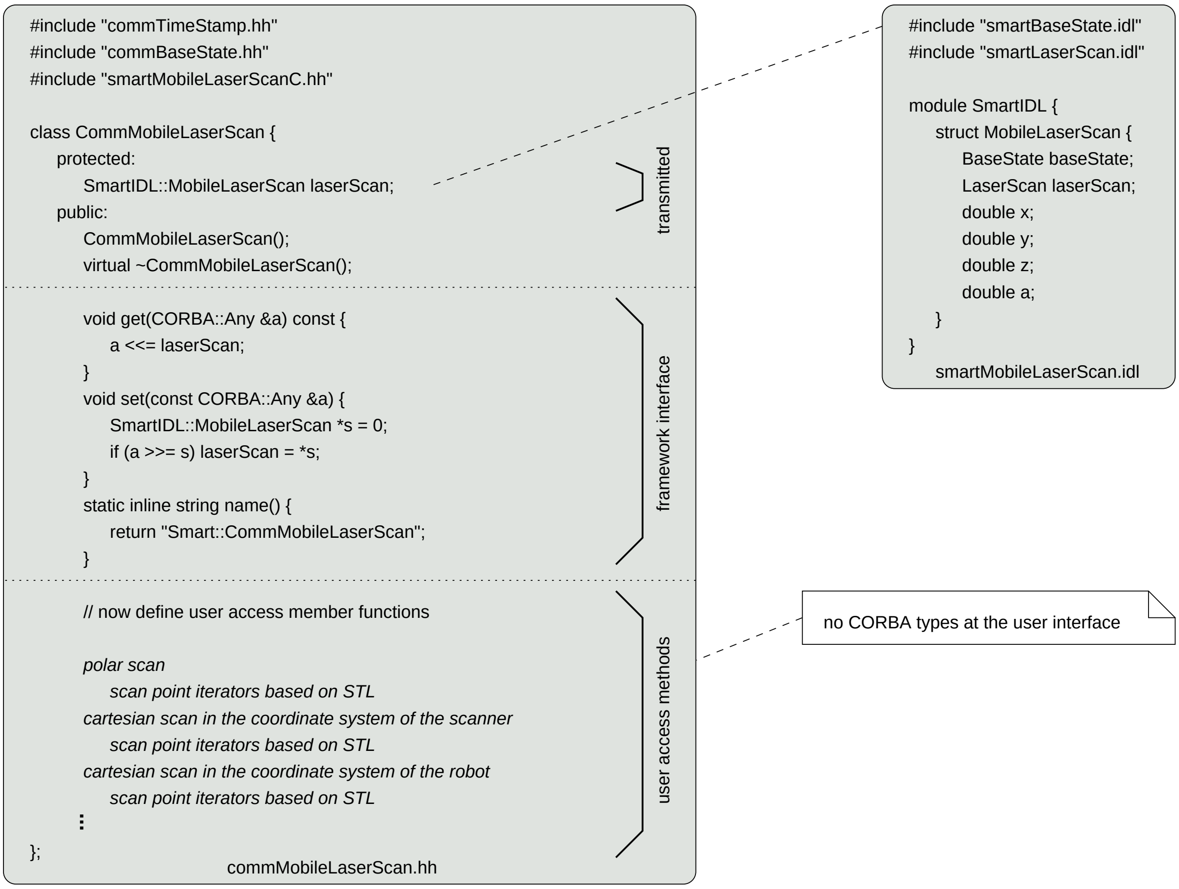
Figure 5.53: *Implementation of the communication object for a time stamp as example.*

the wiring service. By calling *connect(first,laserPort,second,laser)*, it gets connected to the service provider named *laser* of component *second*.

The component which provides the laser scan service is named *second* and is shown on the right side of figure 5.55. The *LaserQueryHandler* implements the handler for incoming requests. The handler instance is converted into an active object by the *QueueQueryServerHandler* decorator. The received requests are stored in a queue which requires only a very small amount of time in the framework executed part of the handler. The thread of the active handler then dequeues one entry after the other and executes the user provided handle member function for every dequeued entry independently of the framework activity.

5.5.5 Summary of the Component Builder View on the Framework

Implementing a new component normally starts with specifying the interface of the component. The interface consists of both provided and needed services. The services to be provided have to be specified with respect to the communication mode and the used communication objects. Normally one reuses communication objects as often as possible to only have one object type per basic entity like laser range scans et cetera. Introducing new communication objects in case that there are no suitable ones available requires only small efforts. Once the required and the provided services are identified and are specified completely including the communication modes and the communication objects,

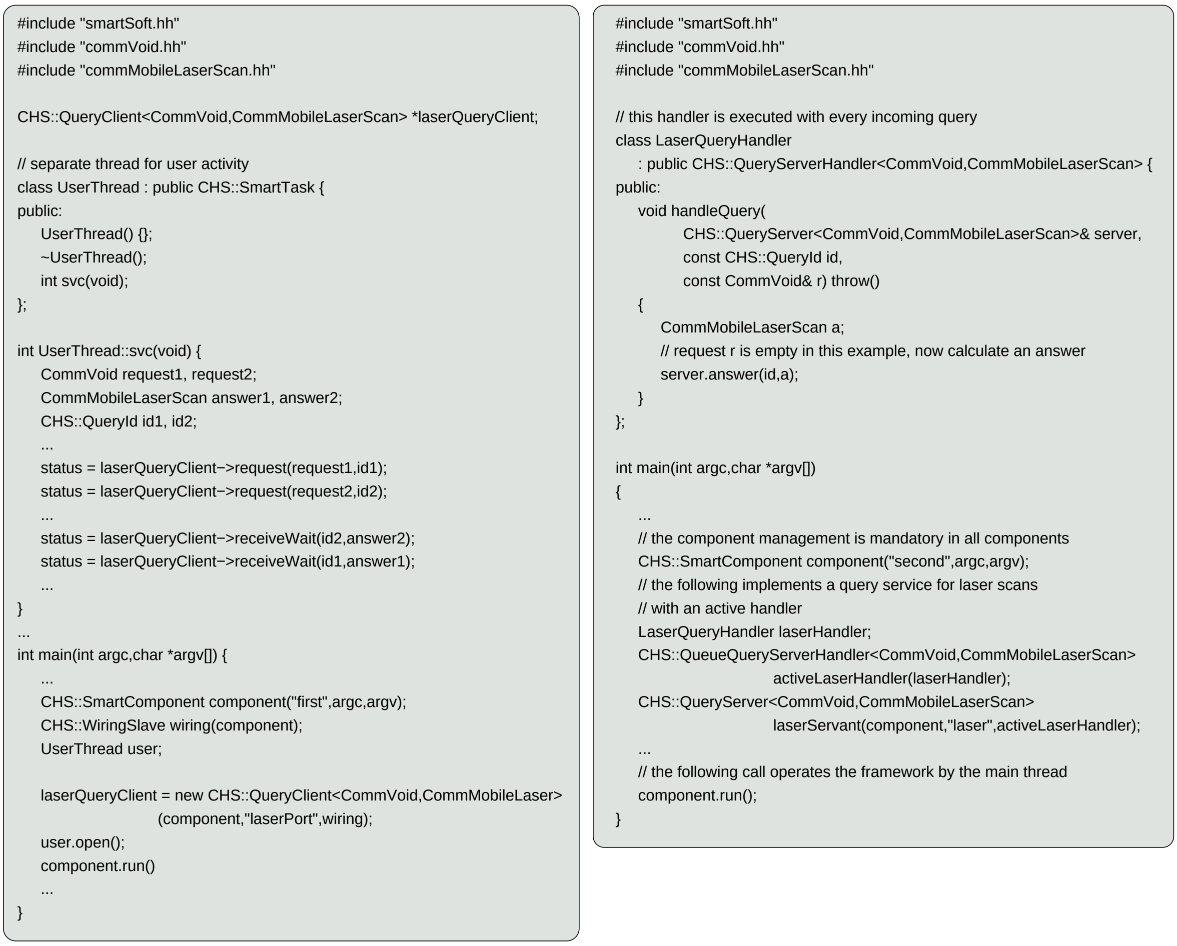


```
#include "commTimeStamp.hh"
#include "commBaseState.hh"
#include "smartMobileLaserScanC.hh"

class CommMobileLaserScan {
    protected:
        SmartIDL::MobileLaserScan laserScan;
    public:
        CommMobileLaserScan();
        virtual ~CommMobileLaserScan();

        void get(CORBA::Any &a) const {
            a <<= laserScan;
        }
        void set(const CORBA::Any &a) {
            SmartIDL::MobileLaserScan *s = 0;
            if (a >>= s) laserScan = *s;
        }
        static inline string name() {
            return "Smart::CommMobileLaserScan";
        }

        // now define user access member functions

        polar scan
            scan point iterators based on STL
        cartesian scan in the coordinate system of the scanner
            scan point iterators based on STL
        cartesian scan in the coordinate system of the robot
            scan point iterators based on STL
        ⋮
};
```

```
#include "smartBaseState.idl"
#include "smartLaserScan.idl"

module SmartIDL {
    struct MobileLaserScan {
        BaseState baseState;
        LaserScan laserScan;
        double x;
        double y;
        double z;
        double a;
    }
}
```

Figure 5.54: *The communication object for a laser scan of a laser range finder that is mounted horizontally on a moving platform.*

one already possesses the complete description of the external interface of the component. Due to the communication patterns, this includes the semantics of the available methods and immediately opens up how to use the interface. In case one only wants to provide an alternative implementation of an already available component, one does not even have to think about the interface but can simply reuse the already available services. One only has to reimplement the handlers of the services to match the new internals of the new component. Implementation of external interfaces is reduced to binding and instantiating templates. Access to remote services consists of calling predefined member functions where one does not have to care about concurrency neither inside the component nor with respect to the service provider outside the component. Providing a service only requires to implement the appropriate handlers of the used communication patterns. The handlers can directly provide the required calculations, can be made active or can forward requests to other threads implementing various processing models. The component builder can focus on the internals of its component. Aspects of other components like their location or how they access the newly provided services are completely removed from the responsibility of the component builder as well as aspects related to the coordination of accessing services. One does not have to care about synchronization neither when accessing services of other components nor within the own component when the own services are accessed concurrently. The component builder however still gets the guarantee that afterwards the new component fits into the already available ones like a piece of a puzzle. The key are the predefined communication



```
#include "smartSoft.hh"
#include "commVoid.hh"
#include "commMobileLaserScan.hh"

CHS::QueryClient<CommVoid,CommMobileLaserScan> *laserQueryClient;

// separate thread for user activity
class UserThread : public CHS::SmartTask {
public:
    UserThread() {};
    ~UserThread();
    int svc(void);
};

int UserThread::svc(void) {
    CommVoid request1, request2;
    CommMobileLaserScan answer1, answer2;
    CHS::QueryId id1, id2;
    ...
    status = laserQueryClient->request(request1,id1);
    status = laserQueryClient->request(request2,id2);
    ...
    status = laserQueryClient->receiveWait(id2,answer2);
    status = laserQueryClient->receiveWait(id1,answer1);
    ...
}
...
int main(int argc,char *argv[]) {
    ...
    CHS::SmartComponent component("first",argc,argv);
    CHS::WiringSlave wiring(component);
    UserThread user;

    laserQueryClient = new CHS::QueryClient<CommVoid,CommMobileLaser>
                        (component,"laserPort",wiring);
    user.open();
    component.run()
    ...
}
```

```
#include "smartSoft.hh"
#include "commVoid.hh"
#include "commMobileLaserScan.hh"

// this handler is executed with every incoming query
class LaserQueryHandler
    : public CHS::QueryServerHandler<CommVoid,CommMobileLaserScan> {
public:
    void handleQuery(
            CHS::QueryServer<CommVoid,CommMobileLaserScan>& server,
            const CHS::QueryId id,
            const CommVoid& r) throw()
    {
        CommMobileLaserScan a;
        // request r is empty in this example, now calculate an answer
        server.answer(id,a);
    }
};

int main(int argc,char *argv[])
{
    ...
    // the component management is mandatory in all components
    CHS::SmartComponent component("second",argc,argv);
    // the following implements a query service for laser scans
    // with an active handler
    LaserQueryHandler laserHandler;
    CHS::QueueQueryServerHandler<CommVoid,CommMobileLaserScan>
                    activeLaserHandler(laserHandler);
    CHS::QueryServer<CommVoid,CommMobileLaserScan>
                    laserServant(component,"laser",activeLaserHandler);
    ...
    // the following call operates the framework by the main thread
    component.run();
}
```

Figure 5.55: *Two example components named* first *and* second*.*

patterns that provide interface methods with fixed semantics and which move arbitrary user definable methods to the communication objects where their scope is restricted to the internals of a component.

5.6 The Framework Builder View on the Approach

The previous section focused on the component builder and described in detail the user interface and the behavior of the communication patterns. This section goes into the details of the internally used mechanisms to achieve the desired behavior of the user interface. It is important to describe not only the user interface but also the protocols and structures behind the user interfaces. Whereas the user interface is mainly important to the component builder, the protocols and structures are relevant to the framework builder. At first glance, the internal mechanisms are not important once the user interface is defined and protocols are often considered as being definable arbitrarily without influencing the user interface. However, slight variations often have significant and non-obvious consequences on the overall behavior and performance. For example, asynchronous user interfaces can become obsolete on top of a synchronous communication mechanism. An unsuited protocol can easily cause a request invocation to return only after the request is fully processed. Therefore, this section presents the pattern internal structures and protocols. These are independent of any specific communication approach and they mediate between the characteristics of the user interface and the characteristics of the communication systems. Details of the interaction of the communication patterns with the underlying communication mechanism are deepened with respect to representative classes of communication systems that already cover a large family of communication systems. The state-of-the-art implementation uses *CORBA* as communication mechanism. Further implementations on top of *TCP sockets*, a *message based system* and synchronous *remote procedure calls* underpin the universality and portability of the approach.

5.6.1 The Role of the Communication Patterns

The communication patterns themselves do not perform the actual communication but mediate between the user interface and the communication mechanism as illustrated in figure 5.56. The communication patterns provide the glue logic which coordinates everything on top of a communication mechanism to enforce the required behavior of the user interface. Communication patterns provide a standardized semantics and behavior independently of the underlying communication mechanism and make sure that the characteristics of the used communication mechanism do not influence the behavior of the user interface. An important demand on the communication patterns is to decouple the service requestor from its service provider including an asynchronous operation of both sides. That is a non-trivial task since not all communication mechanisms provide all of the required features. For example, implementing an asynchronous user interface on top of a synchronous communication mechanism requires carefully chosen protocols and structures. Otherwise, the asynchronous user interface behaves like a synchronous one and becomes obsolete due to the introduced dependencies with respect to execution orders. The challenge is to provide both in parallel, synchronous and asynchronous user interfaces even if only synchronous two-way or asynchronous one-way interactions are supported by the communication system. Both, the implementation of a synchronous two-way user interface on top of asynchronous one-way interactions as well as the implementation of an asynchronous user interface on top of synchronous interactions requires additional glue logic respectively well chosen protocols to achieve the expected user interface semantics.

The only part where the underlying communication mechanism is visible to the component builder is at the framework interface of the communication objects. As already described in section 5.5.3, the communication objects provide framework accessible methods to extract the content of an object for transmission. The way of describing those data structures is normally directly related to the underlying communication mechanism if one uses the marshalling functions of the communication mechanism.

5.6.2 The Interaction Patterns

Communication patterns provide different access modes. The access modes provided at the service requestor and those at the service provider can be used in any combination. From the user level perspective, these combinations of access modes look like different interaction patterns between both parts of a communication pattern. The communication patterns have to implement the interaction patterns independently of the features of the underlying communication system.

The interaction patterns are the core of the communication patterns. They each consist of a service invoking and a service providing part named *client* and *server*. Both parts of a communication pattern, the service requestor and the service provider, can invoke operations on each other and can thus contain both clients and servers of interaction patterns. For example, the client part of the interaction

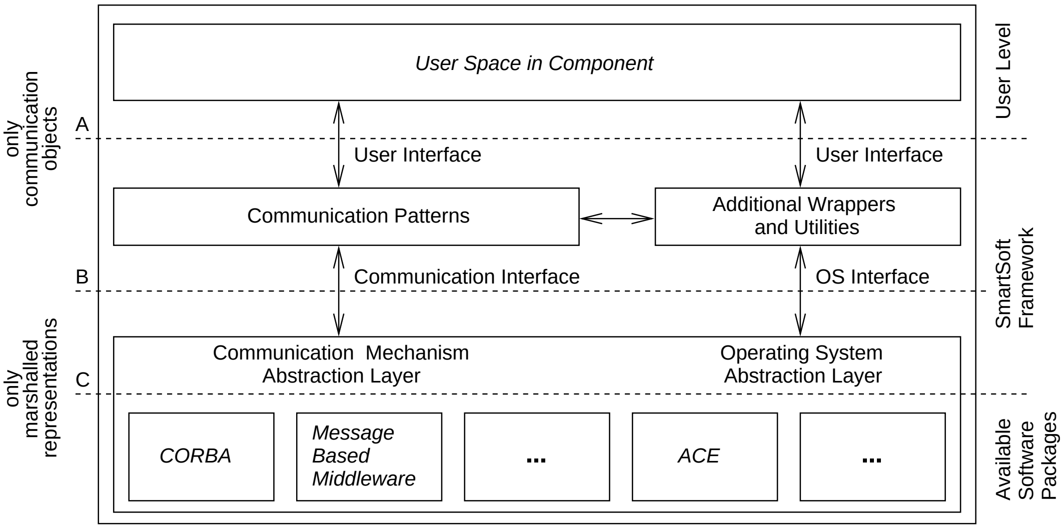

User Space in Component
User Level
only communication objects
A
User Interface
User Interface
Communication Patterns
Additional Wrappers and Utilities
B
Communication Interface
OS Interface
SmartSoft Framework
only marshalled representations
C
Communication Mechanism Abstraction Layer
Operating System Abstraction Layer
CORBA
Message Based Middleware
...
ACE
...
Available Software Packages

Figure 5.56: *The communication patterns provide the glue logic between the user interface and the underlying communication mechanism within components. The dotted stripline labeled with* C *furtheron designates the framework internal interface between the actual communication mechanism and the communication system abstraction.* B *denotes the framework internal interface between the communication system abstraction and the communication patterns. The interface that separates the framework internal level from the user visible part is again labeled with* A.

pattern beneath the *put* method of the server side user interface of the *push* communication patterns is located inside the service provider of the *push* communication pattern.

In principle, one can provide all user access modes with every communication pattern resulting in interaction patterns for all reasonable combinations of client and server side characteristics. However, as already summarized and outlined in table 5.2 in section 5.4.4, not all combinations of access modes and communication patterns are motivated and supported by use cases. Thus, not all possible interaction patterns are currently required by the communication patterns. Nevertheless, the semantics of the user interfaces of the communication patterns and the access modes assignable to the communication patterns are independent of the underlying communication system as soon as all possible interaction patterns are implementable on top of the selected communication system.

5.6.2.1 The Client Part Characteristics

The characteristics of the client part of the interaction patterns are shown in figure 5.57. These characteristics are motivated by the access modes of the communication patterns and can be characterized with respect to the *direction* and the *invocation* mode. The client part interfaces are always based on member function calls. The client part interfaces are expected to be thread-safe, that is no further user level synchronization is needed with concurrent access.

The direction mode can be either *one-way* or *two-way* and determines the transmission direction of arguments. *In* arguments are transmitted from the client to the server and can be modified there but without affecting the client side. An *out* argument cannot provide an initial value to the server but it is returned to the client. An *inout* argument provides an initial value to the server and all modifications are returned to the client. In one-way mode, interactions can have *in* arguments only since there is no back channel and arguments can be transmitted solely from the client to the server. In two-way mode,

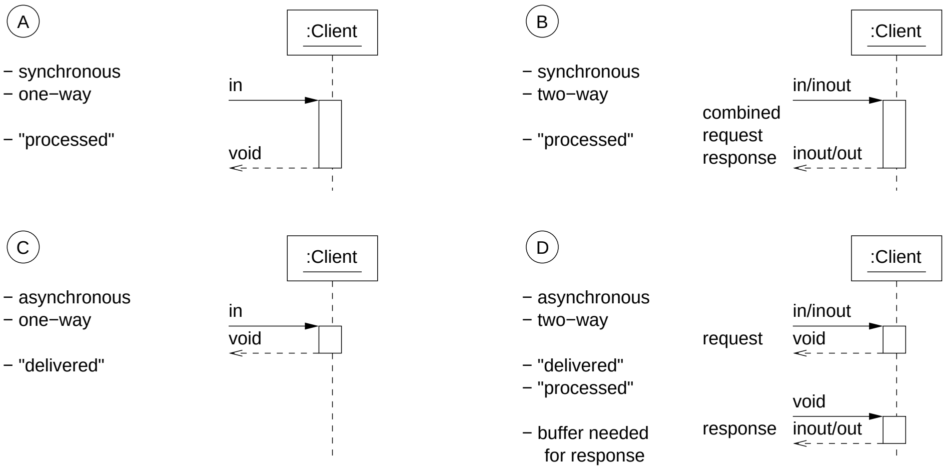


Figure 5.57: *Overview on the characteristics of the client part of the interaction patterns.*

interactions can have all three types of arguments since a two-way interaction provides the required back channel from the server to the client.

The invocation mode can be either *synchronous* or *asynchronous*. A *synchronous* invocation blocks and returns *after* the server side processing is finished according to a *processed* policy whereas an *asynchronous* invocation returns *before*. The invocation mode solely characterizes the client side behavior and does not characterize the communication mechanism used between the client and the server. Asynchronous invocations allow to benefit from concurrent calculations and thus normally result in better reactivity and shorter answer times if calculations can be invoked in parallel.

A client part *synchronous one-way* characteristic (A) accepts *in* arguments only, blocks and returns either after the server side processing is finished or with an error. A feedback channel is still needed to return the *processed* acknowledgment for the synchronous client side interface. However, it is just empty and does not carry any arguments to be returned.

A client part *synchronous two-way* characteristic (B) is invoked with the *in* and *inout* arguments, blocks and either returns with the *inout* and *out* arguments after the server part processing is completed or with an error. The only difference to the synchronous one-way characteristic is that the feedback message is not just empty. The feedback message again corresponds to a *processed* acknowledgment.

A client part *asynchronous one-way* characteristic (C) also transmits *in* arguments only but returns *before* the server part processing is finished or even before it is started. Asynchronous one-way interfaces can be distinguished with respect to the level of guarantees they provide. The *unreliable send* policy returns as soon as the message is delivered to the client part transportation layer and accepts that messages can get lost. The next level is the *reliable send* policy that guarantees the delivery of a message in case the recipient exists. Otherwise, no feedback is given on the whereabout of the message. Even if no message gets lost, one cannot be sure that it is delivered successfully and that it is going to be processed. The recipient might be in the process of destruction or might have disappeared after the message was sent and just before the message is tried to be delivered. Both policies provide no acknowledgment and user level protocols have to take additional precautions to make sure that communicating partners never hold wrong assumptions about the states of their opponents due to messages that never reached their destination. In contrast, the *delivered* policy returns after the message has been delivered to the server part but before it is processed there. However, being delivered guarantees that it is going to be processed and that the recipient cannot get destroyed as long as there are pending requests. Thus, that policy provides an acknowledgment that the message arrived at the server and is going to be processed for sure. The client part *asynchronous one-way* interface (C) is always meant to implement the *delivered* policy.

A client part *asynchronous two-way* characteristic (D) splits the two-way interface into a request and a response method. The request method provides the *in* and *inout* arguments and the response method returns the deferred answer consisting of the *inout* and *out* arguments. The asynchronous two-way characteristic follows a *delivered* policy with respect to the request and a *processed* policy with respect to the response. Since the user decides on when to fetch responses, a buffer is needed to store the answers meanwhile.

The characteristics (A) and (B) and the response method of (D) have to be abortable. Otherwise, one would require the client part to always await the completion of the server part processing before one could react to a changed situation to proceed in a different way. A missing facility to abort those methods would often unnecessarily increase the response time.

In principle, one can emulate all client part characteristics on top of (D). (A) and (B) are emulated by a wrapper that first executes the request method and then blocks by a suitable mechanism until the response can be fetched by the response method. In case of (A), the content of the response is just *void* and carries the acknowledgment that the server part processing is completed. (B) can also directly emulate (A) by restricting the direction mode to *in* arguments only and a *void* back channel. (C) is emulated by using the request method only. Depending on the implementation, however, one might still have to call the response method to close the otherwise still open request. That would make the emulation a bit more complicated since one has to call the not needed response method after the arrival of the *void* response.

The client part of an interaction pattern can always be *passive*. Any activity is initiated from the user level by performing a method call. An upcall from the communication system to the client part never gets blocked since it never performs any further processing besides forwarding the acknowledgment and besides forwarding the answer such that user level threads are released and such that the answer can be picked up via the member function based interface. Both activities do not access any resources whose availability depends on the upcalling thread. Even (B) and (D) are not critical since in case of a successfully invoked request, the administrative structures to accept the response from the communication system upcall are always available. Since each request expects one answer only, there can be no overflow with respect to the administrative structures. In case the answer itself requires too much memory space, it can just be discarded and the administrative structures allow to inform the pending requests on the error without requiring the upcalling thread to wait until enough memory space becomes available. To summarize, the upcall of the communication system never gets blocked in a client part of an interaction pattern.

5.6.2.2 The Server Part Characteristics

The server part characteristics of the interaction patterns are shown in figure 5.58. The server part can be categorized with respect to the *initiation* and the *invocation* mode. The initiation mode can be either *pattern* or *user* and describes the responsibility for initiating the request processing. In *pattern* mode, it is the interaction pattern that makes sure that every incoming request initiates its processing. Thus, one has to consider server part processing models for the upcall to decouple several concurrent requests or to separate the communication from the request processing as detailed in the following section. In *user* mode, it is the user who invokes the processing of requests. Thus, the processing models are not in the scope of the server anymore. However, a buffer is needed for incoming requests that are not yet fetched by the user.

The invocation mode can be either *synchronous* or *asynchronous* and specifies whether a request has to be processed within a single call or is splitted into an invocation and a completion part. A *synchronous* invocation has a single interface method only and interprets the completion of that method as having processed the request. An *asynchronous* invocation is based on an *invocation* and a *completion* interface method. The invocation method provides the *in* arguments and in case of a two-way interaction the *in* parts of the *inout* arguments but does not return any arguments. The completion method has to be called from the user level to provide the *out* part of the modified *inout* arguments and the *out* arguments that have to be returned to the server. Returning from the invocation method to the server does not indicate that the request is already processed. The server considers a request as being completed only after the completion method has been called. On completion, the server can send back any designated arguments respectively a *void* argument. The completion method indicates when a *processed* policy considers the processing as completed.

The advantage of the asynchronous invocation mode is the flexibility with respect to user level processing models. For example, one can easily propagate a request through a pipeline of processing steps and return the result from a thread that is completely different to the one that handled the invocation method. In contrast thereto, a synchronous invocation mode is much easier to implement by the server since it does not require the glue logic to correctly assign provided responses to open requests.

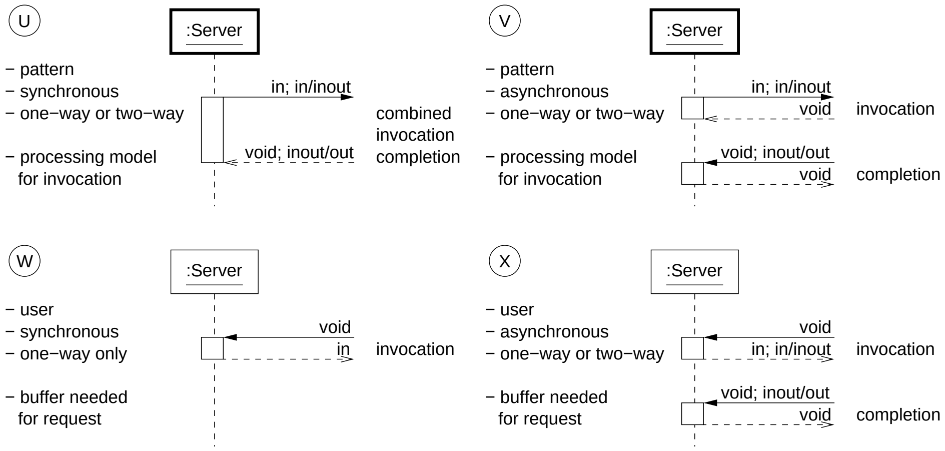


Figure 5.58: *Overview on the characteristics of the server part of the interaction patterns.*

The server part *synchronous pattern* characteristic (U) invokes the processing from the interaction pattern and processes the request within a single upcall. A request is completed with returning from the upcall. The server part *synchronous user* characteristic (W) implements a downcall from the user level to get a request for processing. The actual processing is moved out of the scope of the server and there is no back channel from the user level to the server. Thus, the characteristic (W) is unsuited for interaction patterns that require a *processed* acknowledgment or expect arguments to be returned. The characteristic (W) can be used only in combination with a client part asynchronous one-way characteristic.

The server part *asynchronous pattern* characteristic (V) splits the request into a pattern invoked upcall for the request and a user invoked downcall for the response. Of course, the completion method can also be invoked from inside the invocation method since an asynchronous interface is expected to be organized such that selflocks are impossible. The request is considered as being completed as soon as the completion method is called and independently of any subsequent activities even if these are executed inside the invocation method. In case of one-way arguments, one has to check carefully when to call the completion method since any subsequent activities are not considered as belonging to the scope of the request processing. That behavior is desired with the communication patterns since the user can already complete a request with respect to the client even if there are still some server part housekeeping activities to be executed afterwards. However, some asynchronous interfaces send back an answer earliest after both the invocation method and the completion method returned to the server. Finally, the server part *asynchronous user* characteristic (X) is different to (V) only with respect to the invocation method.

In principle, one can emulate all server part characteristics on top of (V). (W) and (X) require the emulation of the otherwise server provided buffer for incoming requests. The upcalling invocation method of (V) is solely used to put incoming requests into that buffer from where the user can fetch the requests by the emulated invocation method. In case of emulating (W), one still has to call the completion method of (V) just to properly close the otherwise open request. Finally, (U) can be emulated by making the synchronous upcall from inside the invocation method and by calling the completion method after that upcall returns and just before returning to the server.

5.6.2.3 Resulting Combinations of Client/Server Characteristics

An overview on the interaction patterns resulting from all combinations of client and server part characteristics is given in table 5.40. However, not all combinations result in reasonable interaction patterns. The interaction patterns (C/V) and (C/X) are possible but not optimal since the server part response method is not needed with a client part asynchronous one-way characteristic. The completion method at the server still had to be called with *void* arguments to close the otherwise open request. The interaction patterns (A/W), (B/W) and (D/W) are not possible since there is no completion method at the server part. The completion method is needed to indicate the *processed* state as well as to return the *inout* and *out* arguments. The (A/U) interaction pattern is the standard synchronous one-way interaction where the client part returns after the server part processing was finished. The interaction patterns (A/V) and (A/X) would require to call the completion method with *void* arguments after the server part processing is completed. Since the client part characteristic (A) can easily be emulated by (B) and since the server part characteristic (X) is not justified by reasonable use cases, both are furtheron not considered separately anymore. Dropping the server part characteristic (X) and not considering the client part characteristic (A) results in only six different interaction patterns on top of which all reasonable combinations of user access modes of the communication patterns can be implemented. Figure 5.59 shows the remaining interaction patterns.

The interaction patterns (B/U) and (B/V) represent the standard synchronous interactions that can be used in either one-way or two-way mode. The client part returns after the server part processing is completed. In case of (U) this corresponds to the return of the server part upcall and in case of (V) to calling the completion method. The dotted lines correspond to a *delivered* policy with respect to the request and mark the point of time after which the client part invocation has to be abortable. That is in particular important to enable the client part to react before the currently running request is completed since being forced to first await the completion could take far too much time with long running calculations. The completion method of (V) has to be callable from inside the invocation method.

The interaction patterns (C/U) and (C/W) represent the standard asynchronous one-way interaction with either a pattern or a user invoked server part processing and a *delivered* policy. The client part returns before the server part processing is completed or even before it is started but after the

	synchronous pattern U	asynchronous pattern V	synchronous user W	asynchronous user X
synchronous one-way A	✓	✓	— impossible	✓
synchronous two-way B	✓	✓	— impossible	✓
asynchronous one-way C	✓	— not optimal	✓	— not optimal
asynchronous two-way D	✓	✓	— impossible	✓
client part characteristics	server part characteristics			

Table 5.40: *Interaction patterns resulting from all combinations of client and server part characteristics.*

request has been delivered at the server part. Being delivered is tantamount to getting processed for sure.

The interaction patterns (D/U) and (D/V) represent the standard asynchronous two-way interaction. The client part invocation has to implement a *delivered* policy. The server part characteristics (U) and (V) behave exactly like the one in the (B/U) and (B/V) interaction patterns. The client part response methods block if they are called before the response is available and if they are allowed to block. Thus, they have to be abortable.

5.6.3 The Server Part Processing Models

Basic demands on the interaction patterns comprise decoupling and reactivity. Concurrent requests should not block each other and new requests should be accepted even while others are still processed. Both properties are strongly related to the server part threading models. Appropriate threading models allow to accept further requests independently of processing other requests and thus avoid the backpropagation of delays to clients as long as further threads are available. The achievable level of decoupling is directly related to the used threading model.

The server part characteristics (W) and (X) store requests in a buffer until they are picked up by the user level processing. As long as there is enough buffer capacity available inside the server, inserting a request into the buffer does not fail and does not block shared resources. A limited buffer size is sufficient as long as the user level processing model is on average able to pick up and process more requests than arrive. The buffer then compensates for the peak load. Since the actual processing is invoked from the user level, one can do without server part processing models. However, the situation is completely different in case the buffer fills up. Blocking the thread that enqueues new requests or filling up underlying buffers is acceptable unless these are shared resources. Blocking shared resources easily causes deadlocks if the blocked resources are just needed to get through that request that resolves the overload condition. That not only applies to the thread that forwards the request to the buffer but also to shared communication resources.

In contrast thereto, the situation is completely different with the *pattern* initiation modes (U) and (V) where the server is responsible for invoking the request processing. To avoid deadlocks, the user level implementation of the service must not access any resources whose availability depends on the borrowed thread. One can either rely on a server level threading model that already assigns an appropriate and noncritical thread to the invocation method or one forwards the processing to a user level threading model. The latter has the advantage that one is not restricted to the server level threading models since not every communication system already provides the required threading

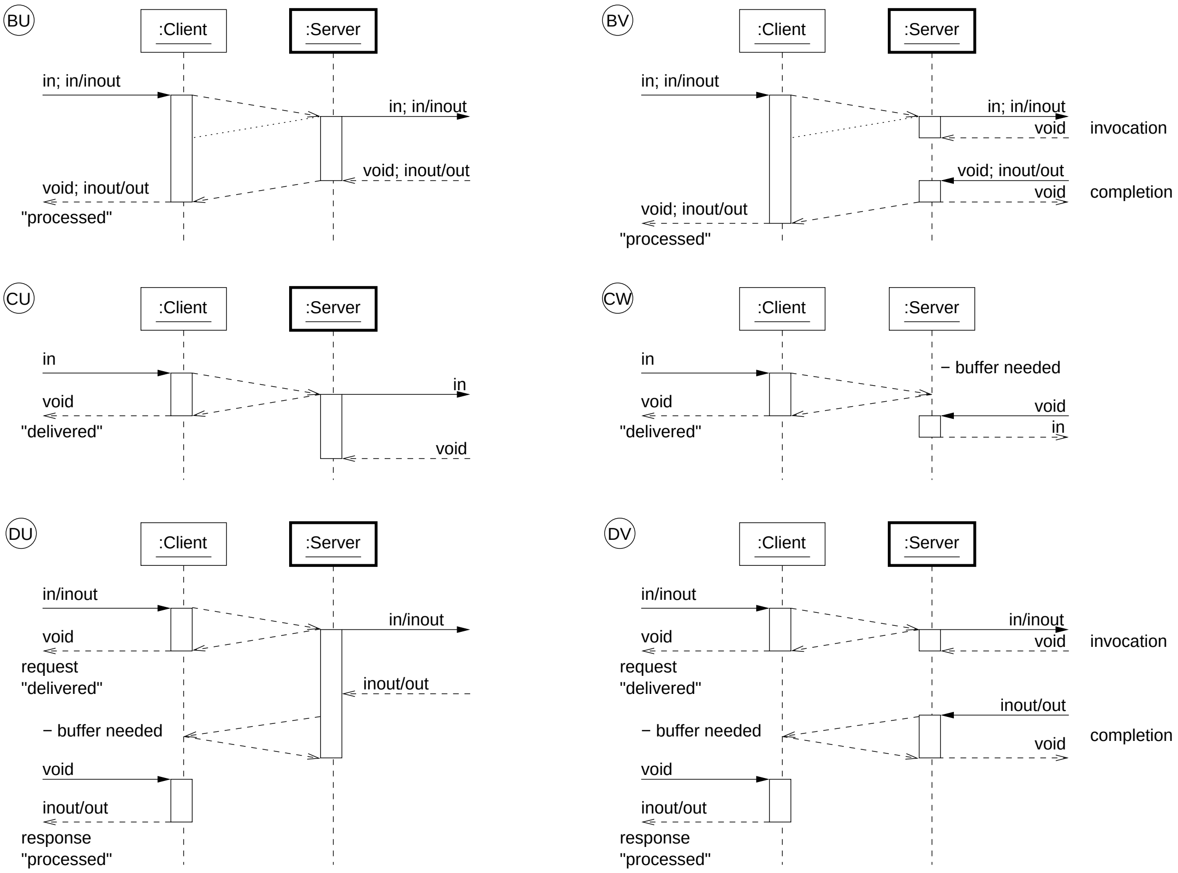


Figure 5.59: *The remaining interaction patterns.*

policies.

Servers can provide different policies to assign threads to requests. For example, a server can invoke a new thread with every request, can have one thread per client or can maintain a thread pool of fixed size. All those models have drawbacks and have to be selected carefully with respect to the application. The *thread per request* model is the only model where a server can handle any number of concurrent requests. Since each server side upcall gets its own thread, its blocking does not prevent another upcall from being processed. The overall behavior is the same as if the original thread was migrated between the client and the server and followed its processing across all communication model boundaries. Thus, each processing sequence that is free of deadlocks (including arbitrarily nested calls) can be executed without having to worry about deadlocks introduced by the distribution. The server behaves like having an unbounded thread pool at its disposal. Since that model can result in a huge number of threads, it can easily reach the limit of a system. In contrast thereto, all threading models that limit the number of concurrently spawn threads at the server can cause deadlocks on an otherwise deadlock-free execution. The execution gets disrupted as soon as no more threads are available and if only another thread would resolve the pending activities. The *thread pool* model results in a deadlock if the maximum number of threads is not sufficient. The *thread per client* model behaves like providing a thread pool of size one for each client. It serializes all requests from one client and the single thread is not sufficient as soon as one request has to pass another request from the same client. That is the case if the server side request processing is blocked on a resource that gets available only after another request from the same client got processed. The *thread per client* model discriminates heavy load clients and the *thread per server* model is like handling all clients with a thread pool of size one. Thus, the requests from all clients are serialized and a deadlock is introduced as soon as the server side processing is blocked on a resource that gets available only after another request from an arbitrary client got processed.

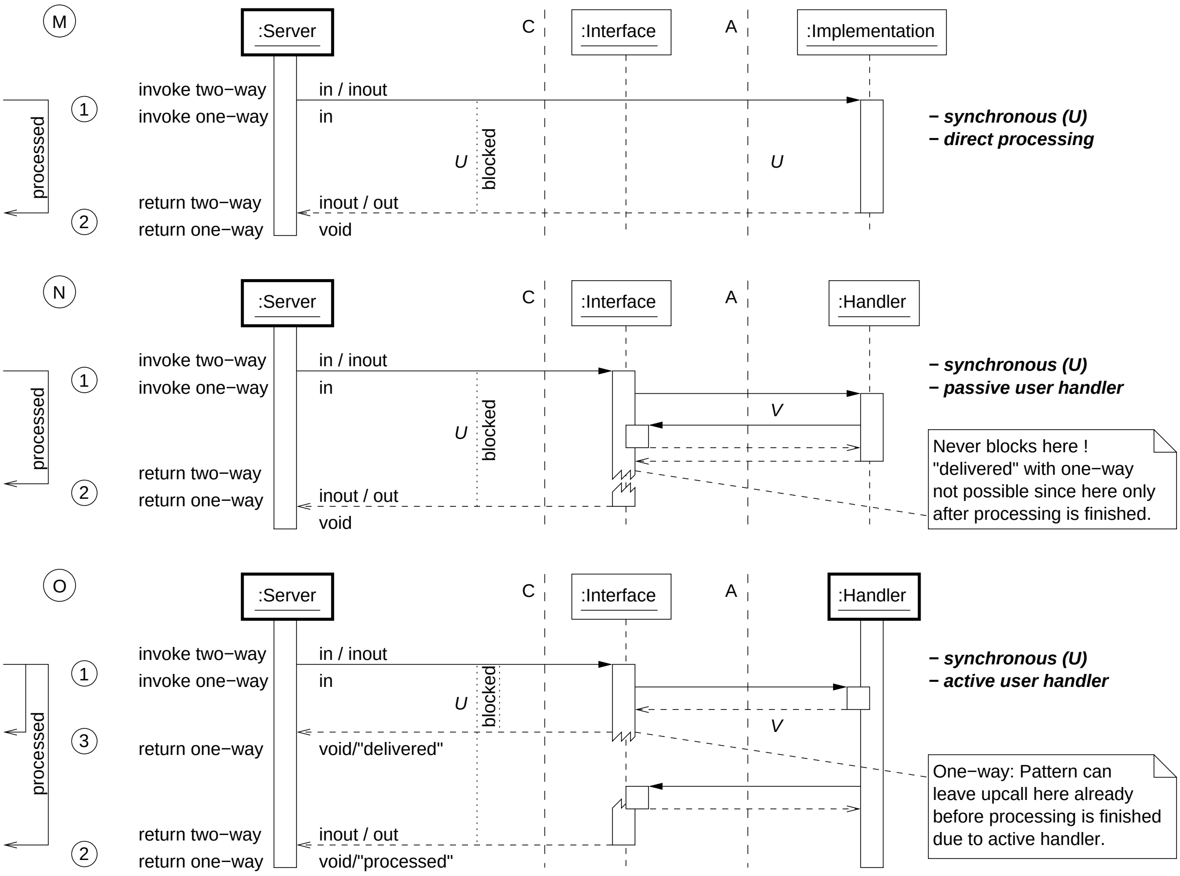


Figure 5.60: *The synchronous server part invocation mode (U) and different processing models.*

Figure 5.60 illustrates the influence of different server part processing models on the interaction patterns in case of a synchronous server part invocation mode (U). The invocation mode (U) declares a request as being *processed* earliest after the upcall from the server to the user level returned to the server. The side-effects of the user level processing on the server and on the interaction pattern are directly related to the used threading model. Likewise are the user level activities restricted depending on the other responsibilities of the thread that invokes the user level implementation. In principle, the upcalling thread must not block on any resources whose availability depends on its own activity.

A server with a *synchronous* server part invocation mode (U) and a *passive direct processing* (M) blocks the thread that invokes the user provided implementation as long as it takes to execute the user provided implementation. This holds true independently of whether arguments have to be returned or not since the upcall from the server returns only after the processing is completed. A synchronous client part invocation returns earliest after ② that is after the server side upcall is completed. In contrast thereto, the client part invocation method of an asynchronous interface with a *delivered* policy always returns latest after the processing is started as marked by ①. Depending on the involved buffers, that might even be before the processing is invoked. Even with one-way arguments only as it is the case with the client part characteristic (A), the synchronous client part invocation cannot be used to directly emulate a *delivered* policy.

A server with a *synchronous* server part invocation mode (U) can forward the request to a separate handler. The synchronous server upcall is split into an *invocation* and a *completion* method as illustrated by the processing models (N) and (O). The implementation of the upcall provides the glue logic to correctly assign returned results to open requests. For example, the implementation of content driven processing chains is much simplified by such an asynchronous interface.

In principle, with an *active* handler, the processing of requests is moved out of the scope of the thread of the server upcall. An active handler separates the request processing from the server upcall. Again, the kind of coupling between different requests of one handler is determined by the chosen handler threading model. The active handler can implement any kind of threading model to assign threads to unprocessed requests. Forwarding a request to an active handler is essential if the required threading policy is not supported by the server. For example, one can order the requests according to priorities or other user specified criteria. An active handler with a buffer can filter out all requests of a particular client and can give them precedence over all other requests. If the handler uses a buffer to store unprocessed requests, one even does not have to await the availability of a thread. As long as there is enough space available in the buffer, one can already return after the request got stored in the buffer and thus even before the processing of the request was started.

An *active* handler (O) on top of a *synchronous* invocation mode does not per se prevent the upcalling thread of the server from being blocked since the synchronous invocation mode has to wait until the active handler provides the results that have to be returned to the server with leaving the upcall ②. Thus, an active handler is not suited to release the upcalling thread of the synchronous invocation mode if a *processed* semantics is required and is also obsolete with two-way interactions. In both cases, an active handler cannot decouple the request processing from the upcalling thread and in both cases, there is no way to circumvent the threading policy of the server upcall. A synchronous client part invocation always results in a *processed* policy and cannot emulate a *delivered* policy.

Generally, the upcalling thread is always occupied from ① to ②. However, it is up to the active handler when to call the completion method that allows the upcall to return to the server. Calling the completion method as soon as possible reduces the blocking time of the server upcall. In case of a synchronous client part invocation, the client part waiting time is reduced in the same order. The client part waiting time then depends only on the point of time of returning from the server upcall. However, only those parts of the handler that are placed before calling the completion method are covered by a *processed* policy. Those parts placed afterwards are still executed but are not covered by the *processed* policy. Thus, the active handler can achieve a *delivered* policy by calling the completion method before doing anything else. That, however, is feasible only with one-way arguments since with two-way arguments, one needs to provide the arguments that are to be returned.

An active handler decouples the server upcall only in case of one-way arguments since one can already return from the upcall after having forwarded the request to the active handler. That, however, requires that both holds true, no arguments have to be returned and a *delivered* policy is sufficient. The blocking time is reduced to the amount of time needed to perform the forwarding. That is much less time than it takes to process the request itself. Even a synchronous client part invocation then already returns after ③. A client part synchronous call then solely covers the forward of the request and the *processed* semantics of the client part invocation represents a *delivered* semantics with respect to the user level processing. That is correct since with an active handler, the processing of requests is moved out of the scope of the server. This has to be considered when interpreting acknowledgments, but it allows to emulate an asynchronous client interface with a *delivered* policy even if only a synchronous interaction with a *processed* policy is available.

Of course, one can do without an active handler. A server with a *synchronous* invocation mode and a *passive* handler (N) completely executes the handler by the thread of the server upcall. The completion method is invoked from the invocation method and thus even before the latter is completed. This requires appropriate mechanisms to avoid selflocks and even in case of an asynchronous one-way semantics, one can not return from the upcall before the processing is completed. The passive handler behaves like the passive direct processing model with the advantage that one can switch to an active handler if that becomes necessary without changing the implementation. A synchronous client part invocation always returns earliest at ② independently of the point of time of calling the completion method since the server side upcall can return only after the thread of the upcall is not needed anymore to execute the handler. The upcalling thread is always occupied until the handler is executed completely. Thus, a synchronous client part invocation always results in a *processed* policy and cannot emulate a *delivered* policy.

To summarize, in case of a synchronous server invocation mode (U) with two-way arguments, the upcalling thread does not return before the request was processed completely. That is independent of using passive or active handlers since one has to await the end of the request processing before one can return from the upcall. The same holds true in case of one-way arguments if one has to fulfill a *processed* policy. Thus, one should make sure that the upcalling thread is not involved in other activities so that it can be used without restrictions to perform the user level processing. An active handler does not make any sense since then the upcalling thread simply gets suspended meanwhile so that it anyhow is not able to perform other activities concurrently to the request processing. If one uses a decoupled thread to perform the upcall, it is just wasted. With a synchronous server invocation mode (U) and two-way arguments, one depends on the threading models of the server. Furthermore, an asynchronous client interface cannot be emulated as long as one uses two-way arguments. With two-way arguments and a synchronous server invocation mode, one needs a communication system that already combines a client side asynchronous interface with a server side synchronous invocation.

In contrast thereto, the upcalling thread of a synchronous invocation mode (U) with one-way arguments can return in case of achieving a *delivered* policy as soon as the request got forwarded to the active handler. Thus, other activities of the upcalling thread are not blocked as long as the active handler is able to accept further requests. Then, one can even share one thread by several servers and the actual assignment of threads to requests is done by the active handlers and independently of the threading models available at the server. The threading models of the server can nevertheless be exploited by using a direct processing model and a passive handler, respectively. However, one then can again not achieve a *delivered* policy since the server assigned thread can be returned only after the processing is completed. Nevertheless, an active handler on top of a synchronous server invocation mode with one-way arguments only is able to emulate an asynchronous client side interface with a *delivered* policy if only a synchronous interaction with a *processed* policy is available.

Generally, there is not a big difference if the processing models are combined with an *asynchronous* server side invocation mode (V) as shown in figure 5.61. The only difference is that the server already provides the split interface and thus hides the glue logic that is otherwise provided inside the communication patterns. Furthermore, the point of time of returning from the invocation method marked by ⑤ has no effect on the client side anymore. The only relevant reference is when the completion method ④ is invoked and thus even a synchronous client part invocation always returns after ④ and independently of the server part processing model. With an asynchronous server side invocation mode, the user gets full control over the point of time at which a *processed* policy is

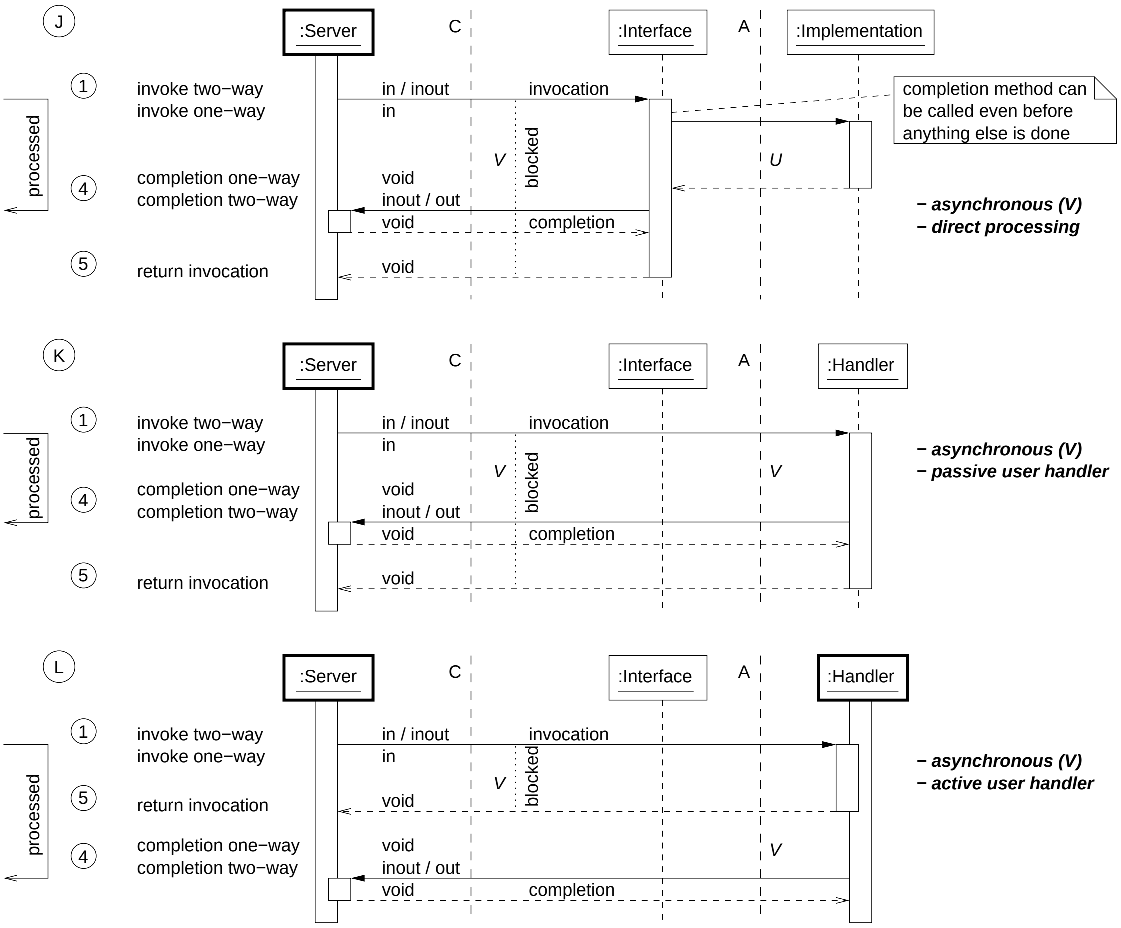


Figure 5.61: *The asynchronous server part invocation mode (V) and different processing models.*

considered as being fulfilled. Independently of the processing models, the *processed* acknowledgment is immediately returned to the client part and not earliest after the upcall was completed. In particular, even with a passive handler, one can reduce the client side delay when awaiting a response by calling the completion method as soon as possible. Further housekeeping activities that, of course, block the upcalling thread in case of (J) and (K), can still be done inside the invocation upcall without affecting the client side. Even with a synchronous client part invocation, the client is not required to wait for that additional time that is required for the processing after having called the completion method. Nevertheless, the point of time of calling the completion method has to be chosen carefully. All activities *prior* to calling the completion method are covered by a *processed* policy. All subsequent activities are still executed but are not covered by the *processed* policy. Calling the completion method *before* doing anything else covers no activities by the *processed* acknowledgment and is thus equivalent to a *delivered* policy. Thus, in case of one-way interactions, one can again emulate a *delivered* policy on top of a synchronous client part invocation that provides a *processed* policy only.

The *passive direct processing* model (J) calls the synchronous user level method from inside the upcalling invocation method, calls the completion method after the user level method returned and then returns to the server. Both handler models, passive (K) and active (L), can be implemented without additional glue logic. All three models, (J), (K) and (L), block the upcall from ① to ⑤.

With a passive handler, the blocking time includes the request processing and with an active handler, ⑤ is already reached after the request got accepted by the handler. Thus, the upcalling thread is blocked only as long as it takes to forward the request to the handler. Since returning from the server part invocation method upcall does not close a request that still has to be kept open, an important difference exists for two-way interactions. These can now take advantage of an active handler. In contrast to the processing model (O), where the upcalling thread is blocked for the time required to perform the processing, the corresponding processing model (L) blocks the upcalling thread only for the time required to forward the request. As a consequence, in contrast to a synchronous server invocation mode (U), an asynchronous server invocation mode (V) can circumvent any server level threading models even with two-way interactions and it is not restricted with respect to implementing user level threading models.

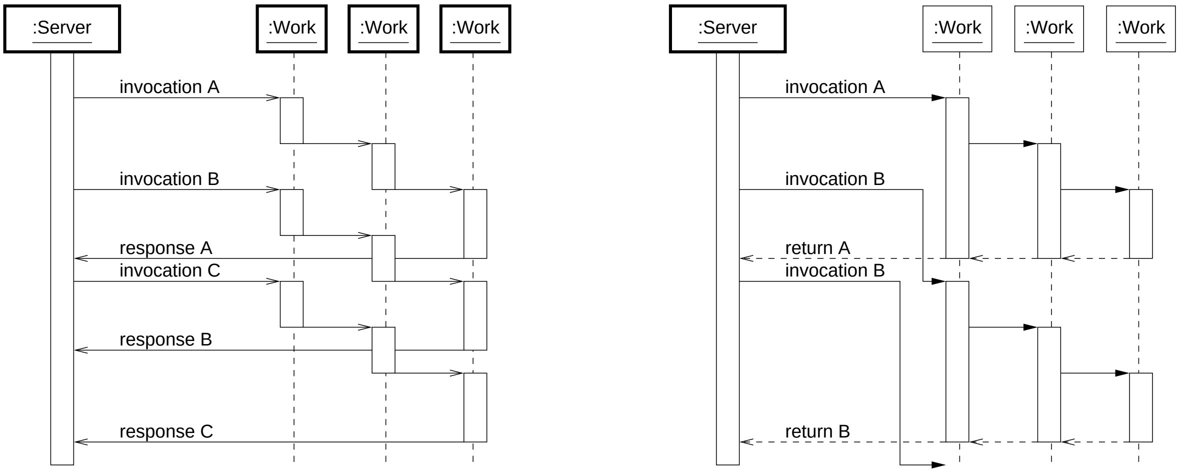


Figure 5.62: *The advantage of an asynchronous server side invocation mode over a synchronous one.*

The advantage of an asynchronous server side invocation mode over a synchronous one is illustrated in figure 5.62. Thereby, there is no relevant difference between the invocation modes (V) and (X). The example shows a request that needs to go through different processing units that can handle only one task at a time. In case of the asynchronous invocation mode shown on the left, one request after the other is pushed through the processing pipeline. Of course, the processing units of the pipeline have to be active, but a single-threaded server is sufficient to exploit the capacity of the processing pipeline. In case of the synchronous invocation mode shown on the right, active processing nodes make no sense since the server upcall always has to await the result before it can return and thus, it is more efficient to use the upcalling thread of the server to invoke passive processing nodes. The overall throughput then, however, is reduced significantly since the first processing unit is blocked until the final result is available and even a multi-threaded server does not improve the throughput due to the synchronous invocation of the processing units. Nevertheless, the throughput can achieve the level of the asynchronous invocation mode if one uses active processing units and if one decouples them from the synchronous invocation according to the processing model (O). A multi-threaded server allows to push further requests into the pipeline concurrently to that ones that are currently being processed. The disadvantage is that one needs as many concurrent server upcalls as there are requests in the processing pipeline. The threads are simply used to await the result that is to be returned and are thus wasted.

To summarize, an asynchronous server side invocation mode provides maximum flexibility with respect to the processing models. An asynchronous invocation mode fulfills all premises to implement any threading model at the user level and one is not restricted to the server provided threading models. The great advantage of an asynchronous server invocation mode is that even with two-way arguments, the upcalling thread gets released as soon as the request got forwarded to an active handler. That is since the invocation and the completion of a request are separated. The deferred delivery of to be returned arguments is already supported by the server interface and does not require the upcall to be blocked as it is the case with the synchronous invocation mode. Therefore, even two-way requests and one-way requests with a *processed* semantics block the server upcall in case of active handlers only for the time needed to forward the request to the handler. Other activities of the upcalling thread are not blocked as long as the handler accepts further requests. In that case, the upcalling thread can be shared by different servers. If the active handler uses a buffer, the upcalling thread even does not have to await the availability of a handler thread and can return immediately after the request got forwarded to the buffer. Again, the threading models of the server can still be exploited by using a direct processing model or a passive handler, respectively.

In principle, threading models and buffers can be placed at various places. They decouple the execution of independent requests and the client part invocation from the server side processing. The interaction patterns have to be implemented carefully to avoid unwanted side effects like blocked servers or wasted resources. Depending on the layer at which threading models and buffers are placed, they cannot be circumvented by user level means without wasting resources. A single-threaded version of a server with an asynchronous invocation mode is sufficient to implement any user level threading model without wasting resources even with two-way interactions. However, in most cases, only synchronous invocation modes are available with common middleware and communication systems. Thus, in case one does not want to waste numerous threads, one is either limited to the provided threading models or one has to use one-way interactions only.

5.6.4 Mapping Interaction Patterns onto Interaction Models

Communication middleware systems provide different interaction models. Typically, not all interaction patterns have a direct equivalent at the level of the communication system. Depending on the capabilities of the communication system, the mapping of the interaction patterns onto the provided interaction models requires varying efforts. Meanwhile, the interaction models of middleware systems like *CORBA*, for example, already cover most of the required interaction patterns so that one could directly map interaction patterns onto interaction models. However, even *CORBA* does not provide an asynchronous server side invocation mode and the memory footage and complexity of highly advanced middleware systems is not suitable for all applications. Generally, as soon as not all interaction patterns have a corresponding interaction model, one needs to appropriately emulate the missing interaction patterns by means of the available interaction models. Independently of the available interaction models, one always has to consider how several interaction patterns of a communication pattern interact with each other and what kind of processing models are required to achieve the desired level of decoupling.

In principle, there are many different ways of mapping the interaction patterns onto the interaction models that are provided by communication systems. Of course, one can individually adjust the implementation of the interaction patterns each time a new communication system is used to eventually gain from the provided interaction models. This might result in a better overall performance but would require significant efforts with every migration. The most obvious disadvantage is that elaborate and careful testing is needed with every migration to ensure the conformance of the implementation with the specification. The implementation of the interaction patterns is in particular demanding and should thus be done only once and inside the communication patterns without requiring adjustments with every migration. Customized adjustments of the mapping of the interaction patterns to fully exploit the individual capabilities of communication systems require a comprehensive understanding of the interplay of the interaction patterns and the interaction models. Therefore, the goal of the explanations in the remaining sections on the framework builder view on the approach is twofold. At first, the relevant details that have to be considered when mapping the interaction patterns onto interaction models and when emulating interaction patterns on top of interaction models are described and motivated. That establishes the basis for individual mappings and optimizations. Secondly, in section 5.6.6, the generic *connection oriented split protocol* is introduced. This protocol alleviates the demands on the communication system such that the interaction patterns can be mapped easily onto most of the mainstream communication systems.

5.6.4.1 Communication Middleware Systems

In general, communication middleware systems can be roughly divided into two large groups with one mainly providing an *asynchronous one-way* and the other a *synchronous two-way* interaction model. Often, the asynchronous one-way interaction is the least common base of message based middleware systems whereas object based middleware systems typically support synchronous two-way interactions. Of course, there are many object based middleware systems that also provide a messaging interface.

Interactions with a *delivered* policy are always considered as implementing a *loose* coupling. The coupling of interactions with a *processed* policy is denoted as *tight* if returning from a client side method depends completely on the server, that is, there are no client side means to abort an interaction. Typically, a *tight* coupling between the client and the server is implemented by synchronous communication mechanisms if no support from the communication system is provided to abort blocking calls.

interaction model	client characteristics				server characteristics			
E	C	asynchronous	one-way	delivered	W	synchronous	one-way	user
F	B	synchronous	two-way	processed	U	synchronous	two-way	pattern
G1	C	asynchronous	one-way	delivered	U	synchronous	one-way	pattern
G2	-	asynchronous	one-way	reliable send	U	synchronous	one-way	pattern
G3	-	asynchronous	one-way	unreliable send	U	synchronous	one-way	pattern
H	D	asynchronous	two-way	delivered	U	synchronous	two-way	pattern

Table 5.41: *Summary on interaction models supported by communication middleware systems.*

Figure 5.63 illustrates the interaction models that are typically supported by communication middleware systems and table 5.41 summarizes their main characteristics. Furtheron, an asterisk indicates that a client side method is not abortable. In case of (F)*, that is related to the method which performs the synchronous interaction and in case of (H)* to the method that awaits the answer.

The interaction models (E), (F), (G1) and (H) correspond to the interaction patterns (C/W), (B/U), (C/U) and (D/U). The other interaction models do not have a direct equivalent. The interaction model (G2) provides a *reliable send* policy, that is no message gets lost but no feedback is given if a message is dropped due to a nonexisting recipient. The interaction model (G3) even allows messages to get

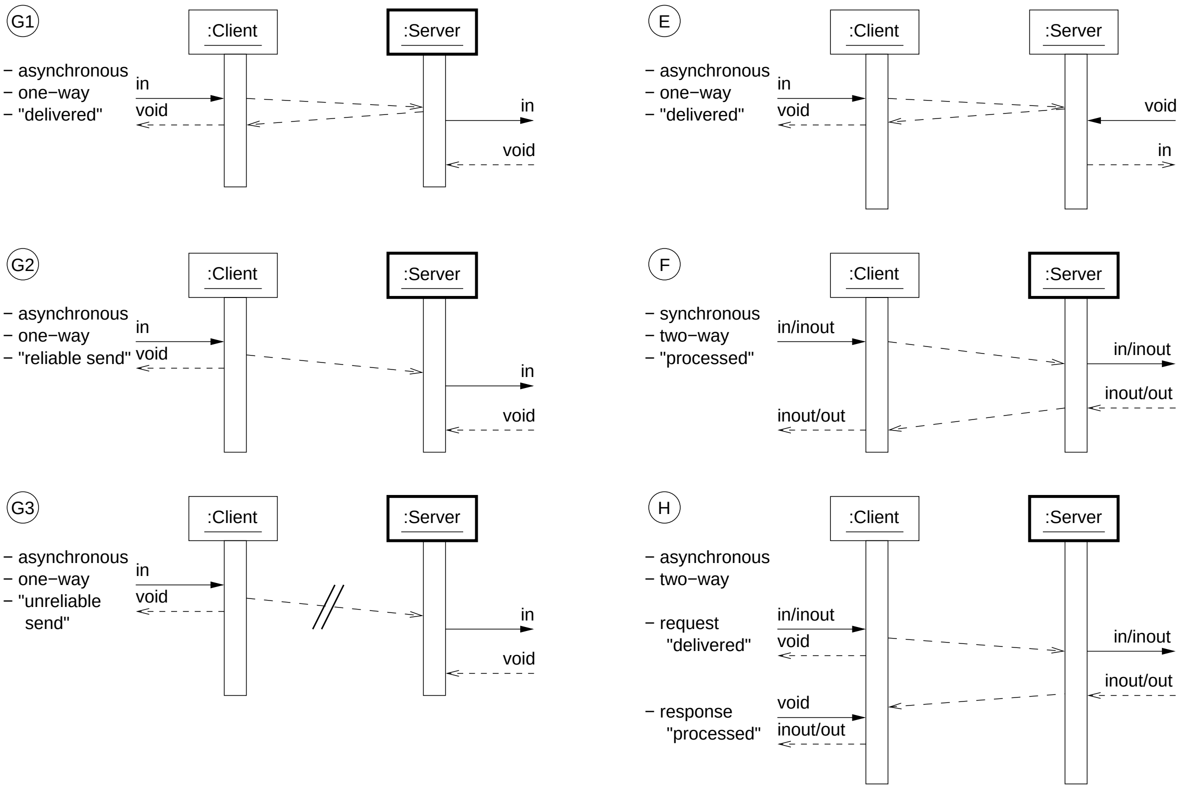


Figure 5.63: *Overview on interaction models supported by communication middleware systems.*

lost and is not considered here.

Most message based systems implement the interaction model (G2) providing a *reliable send* policy where incoming messages invoke a before registered upcall. Often, the only way to implement two-way interactions is to split them into two independent one-way messages which requires appropriate glue logic to correctly compose the independent interactions. Additionally, message based systems typically support single-threaded servers only and thus require user level processing models to avoid a blocked server. Of course, the advantage is that one can implement any user level processing model without wasting server resources. Another advantage of message based systems is their communication level asynchronicity that already decouples the sender and the receiver according to a loose coupling.

TCP sockets behave like the interaction model (E). One needs at least one thread that is always able to accept incoming data to avoid that operating system buffers get blocked and that delays get propagated back to the sender. Threading models are essential to decouple the user level activities from the communication activities.

Some message based systems are based on *mailboxes*. In contrast to a socket based approach, mailboxes already provide the buffers for incoming messages. Since they also behave like the interaction model (E), one again needs at least one thread to pick up and dispatch the mailbox entries. Again, threading models are essential to decouple the dispatching of mailbox entries from the user level activities.

Remote Procedure Calls (RPC) typically implement the interaction model (F)*. The difficulty is to implement the asynchronous interaction patterns if only synchronous interactions are provided and to achieve the required loose coupling. Very often, *RPC* implementations neither support asynchronous client side invocations nor advanced threading models.

Many object based middleware systems support various interaction models, always including the interaction model (F)* as standard interaction model. (F)* also describes the standard remote method invocation of *CORBA*. Due to the synchronous server side interface, object based middleware normally supports various object level threading models. Of course, this is not only to avoid serialization of requests and to avoid blocking the communication activities but also to relieve the middleware user from resource allocation problems as far as possible. The price to pay is being restricted to the provided threading models or wasting resources when additionally implementing user level threading models. With *CORBA*, for example, more elaborate threading models like priority based thread assignments are specified in the *CORBA 3* realtime extensions only. These are supported by very few *CORBA* implementations only, impose further demands on the overall system and are therefore an overkill for most applications. Unfortunately, *CORBA* does not support asynchronous servers and one thus has to live with the drawbacks of synchronous two-way server side interfaces.

CORBA provides *oneway* declarations for methods that have *in* arguments only [134]. The client side invocation can be completed by the *object request broker (ORB)* before the server side processing is finished since no return arguments have to be awaited. Therefore, *oneway* declarations change the client side characteristics of methods with *in* arguments only from *synchronous one-way* to *asynchronous one-way*. However, the *CORBA 2.x* specification does not include any guarantees for *oneway* declared methods and thus the reliability of the interaction depends on the features of the used *CORBA* implementation. That has been recognized as severe drawback and at least *CORBA 3* provides several policies for *oneway* declarations called *sync scopes* [120]. These now provide the already described guarantees for asynchronous interactions. For example, the *sync none* policy returns immediately after the request got forwarded to the local *ORB* but even before the request is passed to the transportation layer and thus corresponds to (G3). In contrast thereto, the *sync with server* policy returns after the request was delivered at the server side *ORB* but before it is processed there. However, being delivered guarantees that it is going to be processed and that the recipient cannot get destroyed as long as there are pending requests. Thus, that policy provides an acknowledgment that the request arrived at the server and is going to be processed for sure and thus corresponds to (G1). Only the newly introduced *sync with server* policy turns the unreliable *CORBA 2.x oneway* interactions into reliable asynchronous one-way interactions with a *delivered* policy.

Finally, the *asynchronous messaging interface (AMI)* of *CORBA* implements an asynchronous two-way interaction. The goal of the *AMI* is to provide an asynchronous client side interface without requiring any server side changes. Two-way interactions are automatically split into a one-way request method and a data structure to hold the lateron received response. The standard synchronous remote methods get usable in an asynchronous way. However, the *AMI* results in bulky interfaces since the *CORBA* stub always contains both the method for the synchronous invocation and the methods and structures needed for the asynchronous invocation. The *AMI* comes in two flavors, namely the *polling* and the *callback* model [5]. From the perspective of the user, the polling model is easier to handle since the user can decide on when to call the response method and since the client side buffer is provided by the communication system. However, the polling model is not as efficient as the handler based version. In contrast to the polling model, the handler based version requires more elaborate user level structures to properly synchronize deferred activities with received responses and it requires user level buffers. The polling *AMI* model corresponds to the (H)* interaction model since the client side request implements a *delivered* policy and since the response provides a *processed* semantics.

The advantage of the *CORBA reliable oneway* approach is the communication level asynchronicity that already decouples the client stub and the servant by means of the *ORB* and its buffers. Thus, one does not have to achieve that decoupling at the framework level. The *ORB* level decoupling is also the basis of the *CORBA AMI* model that enriches the client side interface by asynchronous invocations. It then leaves it to the user to invoke a remote method either synchronously or asynchronously. However, in case of the *AMI* model, one still has to live with the drawbacks of synchronous two-way server side interfaces and in case of the *reliable oneway* approach, one can transmit arguments into one direction only.

Unfortunately, asynchronous communication mechanisms are either only available with message based systems that typically provide one-way interactions only or are only available as client side invocation model with a synchronous server side invocation as it is the case with some object based middleware systems. Apparently, nearly all communication middleware systems provide a server side synchronous characteristic only. At least with two-way requests and with one-way interactions with a *processed* semantics, this results in wasted resources if one does not stick to the server level threading models.

For all of the above communication middleware systems, messages or requests between a particular client and a particular server keep their initial order and never pass each other. Since that holds true for most of the communication middleware system or can be ensured by the lower layers inside a communication middleware system, it is furtheron assumed that this property always holds true. With respect to the interaction patterns, the kept order simplifies the protocol that emulates interaction patterns on top of various interaction models.

5.6.4.2 Potential Mapping Options

The goal is to find a generic mapping between interaction patterns and interaction models such that the modifications that are required in case of migrating to another communication system are reduced to a minimum. Mapping the interaction patterns onto interaction models that form the least common basis across typical communication systems would require adjustments at the syntactical level only but not at the level of the communication logic inside the communication patterns.

required interaction patterns	provided interaction models							
	CORBA		*RPC*		message based		*TCP sockets* and *mailboxes*	
(B/U)	F*	(B/U)	F*	(B/U)	G2	—	E	(C/W)
(B/V)	G1	(C/U)						
(C/U)	G3	—						
(C/W)	H*	(D/U)						
(D/U)								
(D/V)								

Table 5.42: *Required interaction patterns and provided interaction models.*

A summary on the required interaction patterns and the interaction models typically provided by the various types of communication middleware systems is given in table 5.42. As already mentioned, at least either a synchronous two-way (F)* or an asynchronous one-way (G1, G2, E) interaction model is provided. Again, the asterisk marks missing facilities to abort blocking calls. However, none of the interaction models is available across all communication systems, neither the synchronous two-way nor the asynchronous one-way interaction model.

A passable approach is to emulate the interaction patterns on top of a single interaction pattern only. The advantage is that this approach achieves a very lean interface between the communication patterns and the communication system with only one interaction pattern being visible. All other interaction patterns are emulated and are thus not affected by a migration and in case of migrating to another communication system, one has to reimplement one interaction pattern only. Furthermore, the conformance checks have to be performed with one interaction pattern only. That interaction pattern, however, has to be able to emulate all other interaction patterns and it has to be mappable onto different interaction models. In particular, it has to work on top of a synchronous interaction pattern as well as on top of an asynchronous one, either one-way or two-way. Those requirements alleviate the demands on the communication system such that they can be met by nearly any of the mainstream communication systems.

Since the (D/V) interaction pattern can emulate all other interaction patterns, one option is to solely map the (D/V) interaction pattern onto one of the interaction models provided by the used communication system and to emulate all other interaction patterns by means of the (D/V) interaction pattern. Unfortunately, the (D/V) interaction pattern cannot be mapped easily onto the typically available interaction models. Actually, it often requires extensive and complex glue logic.

In contrast to the (D/V) interaction pattern, the (C/U) interaction pattern is much easier to implement on top of standard interaction models. The first part of the (D/V) interaction pattern already consists of a (C/U) interaction pattern and the second part is a (C/W) interaction pattern. The characteristic (W) is emulated on top of the characteristic (U) by a buffer that is filled by the upcall of (U) and that is read by the interface method of (W). The interface of the communication patterns to the communication system is again reduced to a single interaction pattern only. However, the (C/U) interaction pattern is mapped onto standard interaction models much easier.

5.6.5 The Split Protocol and its Challenges

The leaner solution compared to the two-stage emulation of the interaction patterns on top of the (D/V) interaction pattern that is emulated by two (C/U) interaction patterns is to directly emulate the interaction patterns by solely using the (C/U) interaction pattern. That approach is called the *split protocol* approach. Since the (D/V) interaction pattern can already emulate all other interaction patterns and since that can be emulated by the (C/U) interaction pattern, the (C/U) pattern is also sufficient to emulate all other interaction patterns.

Synchronous interactions and two-way interactions are split into two independent one-way interactions. The invocation message transmits the *in* arguments and the *in* part of the *inout* arguments from the client to the server and the completion message transmits the *out* part of the *inout* arguments and the *out* arguments from the server to the client. In case of emulating a synchronous one-way interaction, the completion message is just empty and provides the *processed* acknowledgment only.

The main advantage of the split protocol approach again is the lean interface between the communication patterns and the communication system. The (C/U) interaction pattern is the only one that is visible at the interface of the communication patterns to the communication middleware system and that has to be mapped onto an interaction model. It is important to note that there are two features of the (C/U) interaction pattern that predestinate it as interface pattern. Due to the one-way characteristic, it can be mapped onto communication systems that provide a one-way interaction only and due to the *delivered* policy, it already provides the required decoupling between the client and the server part as required by the interaction patterns.

In particular, the *delivered* policy is mandatory and is exploited by the emulation of the other interaction patterns. Even if it looks like the (C/U) interaction pattern could be mapped easily onto all interaction models, it is challenging to achieve the *delivered* policy on top of the various interaction models. On top of a synchronous interaction model, one has to convert the *processed* policy into a *delivered* policy without ending up at a *reliable send* policy only. On top of a *reliable send* policy, one has to achieve the guarantee of the *delivered* policy that requests get processed once they are delivered successfully.

This section illustrates those challenges and discusses several solutions and their drawbacks as preparation for the *connection oriented split protocol* introduced in section 5.6.6.

5.6.5.1 Emulating Interaction Patterns by the (C/U) Interaction Pattern

The basic interface between the communication patterns and the underlying communication mechanism is shown in figure 5.64. *B* denotes the interface of the communication patterns to the communication system abstraction. The emulation of interaction patterns is part of the communication patterns. Communication patterns provide and accept user level data in marshalled form only. Messages are sent by invoking the client part of an appropriate (C/U) interaction pattern and messages are received via callbacks that are invoked by the upcalling server part of a (C/U) interaction pattern. Due to the one-way semantics of the split protocol, neither the methods called to send data can return any parameters from the recipient nor can the invoked callbacks return any parameters to the sender. Of course, the split protocol requires appropriate glue logic inside the communication patterns to coordinate the independent (C/U) interactions. The price to pay is the increased complexity of the implementation of the interaction patterns since these have to be emulated on top of the (C/U) interaction pattern inside the communication patterns. However, that additional complexity has to be mastered only once with the implementation of the interaction patterns and not with every migration to a new communication middleware system. It is much better to cope with these demanding details only once and inside the communication patterns rather than with every middleware migration. The additional complexity of the communication patterns that host the emulation of the interaction patterns thus pays off very soon.

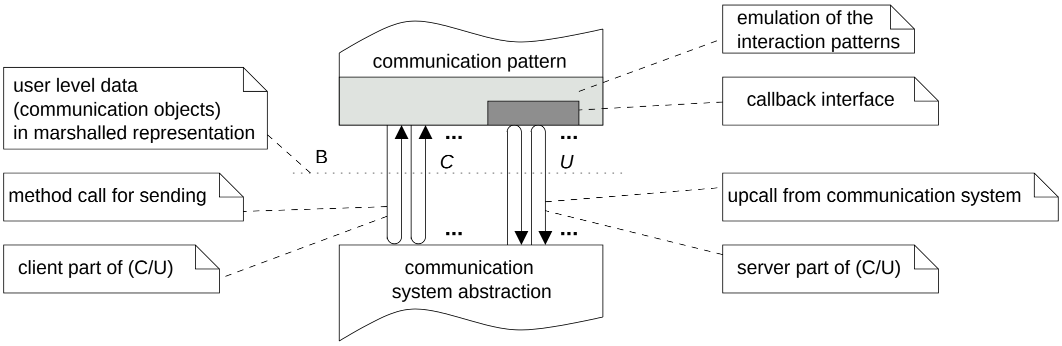

emulation of the interaction patterns
communication pattern
user level data (communication objects) in marshalled representation
callback interface
B
C
U
...
method call for sending
upcall from communication system
client part of (C/U)
server part of (C/U)
communication system abstraction

Figure 5.64: *The interface between the communication patterns and the underlying communication mechanism.*

Figure 5.65 illustrates the split protocol on top of the (C/U) interaction pattern. The simplified illustration of this interaction is shown in figure 5.66. *D* is the invisible intercomponent interface hidden by the communication mechanism. The interaction is initiated from the client part shown on the left and it works exactly the same way vice versa.

Table 5.43 gives an overview on emulating the interaction patterns by means of the (C/U) interaction pattern. The independent (C/U) interactions of the split protocol already provide an asynchronous server side invocation mode (V) that is otherwise only hard to achieve. Emulating a synchronous characteristic is the easier part even if it requires substantial glue logic. Using multiple interaction patterns inside a communication pattern results in the appropriate number of independent (C/U) interaction

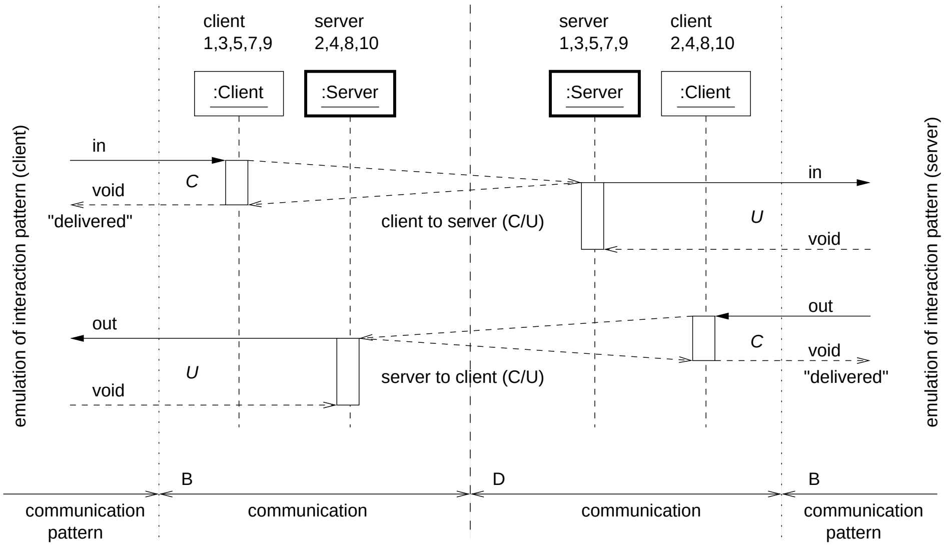


Figure 5.65: Implementing interaction patterns on top of (C/U) interaction patterns.

patterns.

The client side synchronous characteristic of the (B/U) interaction pattern is emulated by a wrapper that first invokes the client part of ① and then blocks by a suitable mechanism until the response arrived by ②. The advantage of the split protocol is that the client side blocking that awaits an answer is handled locally and can thus be implemented such that it can be aborted locally as required by the blocking flag without any further support from the communication system. The server side characteristic is emulated by calling a synchronous user level method from inside the upcall of ①. The client part of ② is invoked from inside the upcall of ① after the user level method returned and just before leaving that upcall.

The client side of the (B/V) interaction pattern is the same as for the (B/U) interaction pattern. However, the client part of ④ is now invoked from inside the server side completion method.

The (C/W) pattern requires one (C/U) interaction pattern only. The client side interface method of the (C/W) pattern invokes the client part of ⑤ and the server side upcall of ⑤ forwards the request to a buffer.

The client side request method of the (D/U) interaction pattern maps directly onto the interaction ⑦ and the emulation of the server side characteristic (U) is the same as for the (B/U) interaction pattern. The upcall of ⑧ at the client side of the (D/U) interaction pattern is used to forward the response to a buffer.

The first part of the (D/V) interaction pattern maps directly onto the interaction ⑨. The server side completion method invokes the client part of ⑩ and the upcall of ⑩ at the client part of the (D/V) interaction pattern again forwards the response to a buffer.

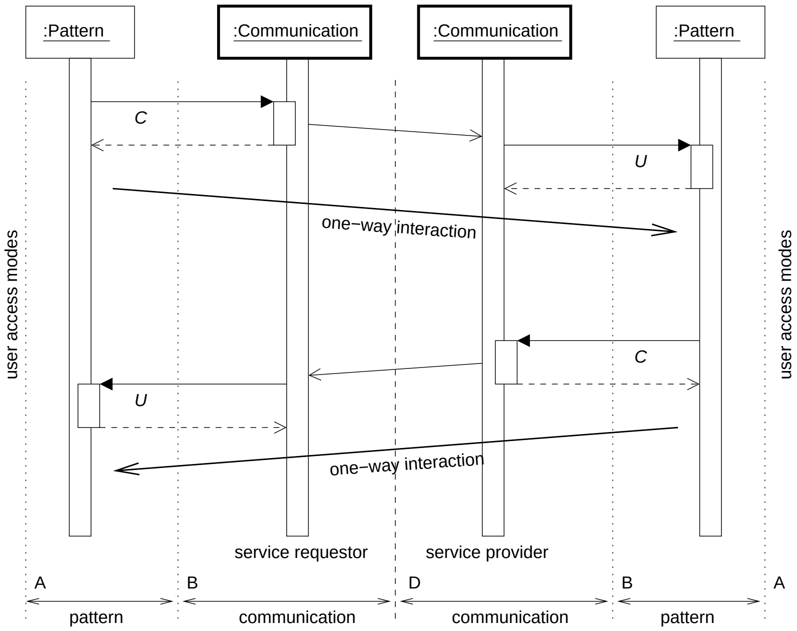


Figure 5.66: *The basic structure of the interface of the communication patterns to the underlying communication mechanism. The pattern to pattern interaction is based on one-way messages with a* delivered *policy.*

	(B/U)	(B/V)	(C/W)	(D/U)	(D/V)
(C/U) interactions	2	2	1	2	2
user interface client	wrapper	wrapper	direct	request: direct response: blocks on buffer	request: direct response: blocks on buffer
user interface server	wrapper	direct	blocks on buffer	wrapper	direct
client to server	①	③	⑤	⑦	⑨
client side downcall	inside wrapper	inside wrapper	direct	request method	request method
server side upcall	execute user level method, send answer from inside upcall *after* processing is completed	execute *invocation* method	forward to buffer	execute user level method, send answer from inside upcall *after* processing is completed	execute *invocation* method
threading policy	relevant inside emulation beneath user interface	sufficient at user level	relevant inside emulation in case of filled up buffers	relevant inside emulation beneath user interface	sufficient at user level
server to client	②	④	-	⑧	⑩
server side downcall	(see above)	called by *completion* method	-	(see above)	called by *completion* method
client side upcall	signals wrapper	signals wrapper	-	forwards to buffer, signals *response* method	forwards to buffer, signals *response* method
threading policy	not required	not required	-	not required	not required

Table 5.43: *Emulating the interaction patterns by two independent (C/U) interactions.*

5.6.5.2 Processing Models and the Emulation of the Interaction Patterns

Figure 5.67 summarizes the processing models and their effects on the split protocol. Of course, the upcalling thread at the server part of the (C/U) interaction pattern is blocked from ① to ⑦. Due to the one-way arguments of the upcalls from the communication system, one can already return to the server as soon as the request got forwarded and due to the split protocol, returning to the server does not implicitly signal a *processed* acknowledgment. Generally, threading models are relevant at the server part of the (C/U) interaction pattern if the upcall blocks on resources that get available only after the upcalling thread is released. If the upcall requires only resources that are available independently of the upcalling thread, one can use the upcalling thread to perform the processing.

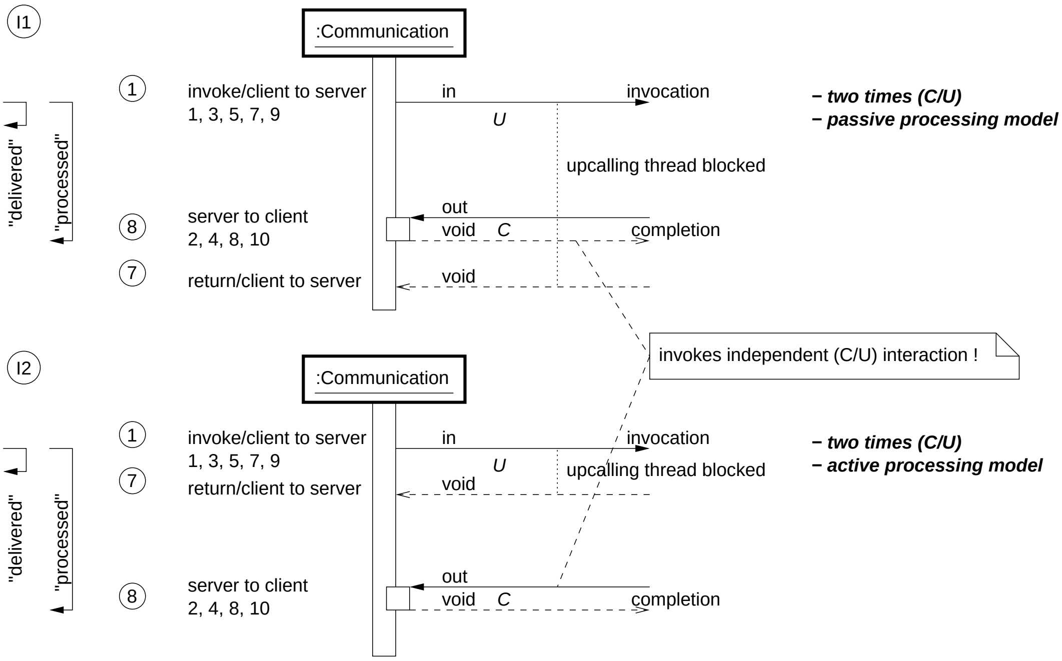


Figure 5.67: *The (C/U) split protocol and processing models.*

The upcalls of ③ and ⑨ shown in table 5.43 invoke the user level interface at the server side of the interaction patterns (B/V) and (D/V), respectively. Thus, the user level implementation can block the upcalling thread that might be needed to keep the interaction pattern, the communication pattern or the component alive. The important point is that user level threading models are sufficient to achieve any level of decoupling between the upcalling thread of the interaction pattern and the user level activities. With a user level threading model, one does not need any threading model inside the interaction pattern and one does not waste threads in combination with an active user level processing model. Nevertheless, one can exploit threading models provided by the communication system due to the decoupling of the *delivered* policy of the (C/U) interaction pattern. If the threading models provided by the communication system are sufficient, one can relinquish the thread assignment to the communication system and one can use passive user level processing models only.

In contrast thereto, user level threading models at the server cannot decouple the upcalling thread of ① and ⑦ from the user level implementation. The only way is to already use a separate thread per invocation of the emulated synchronous user level interface. The situation is the same as with the processing model (J) in figure 5.61. The invocation method compares to the upcall of the (C/U) interaction pattern and the completion method to invoking the client part of ② and ⑧, respectively. Since returning from the emulated synchronous user level interface is always interpreted as closing the processing of the request, one cannot take advantage from a user level threading model. One still has to await the completion of the processing as it is the case with the processing model (O). Otherwise, ② and ⑧ would be invoked too early and would thus violate the *processed* characteristic of the (B/U) and the (D/U) interaction pattern. Since the upcalls of ① and ⑦ return only after the user level method was executed, the upcall is blocked from ① to ⑤ and independently of the user level processing model. To summarize, the emulated synchronous user level upcall depends on the threading model beneath the user level interface. Since all resources needed by the user level implementation must be accessible without first releasing the upcalling thread and since one should not restrict the user level to certain resources only, one always needs a separate thread per user level upcall. In case the *thread per request* model, that assigns a separate thread to every upcall, is not available with the server part of the interactions patterns ① and ⑦, one needs a separate thread per user level upcall inside the server part emulation of the (B/U) and the (D/U) interaction pattern, namely above the communication system and beneath the user level interface.

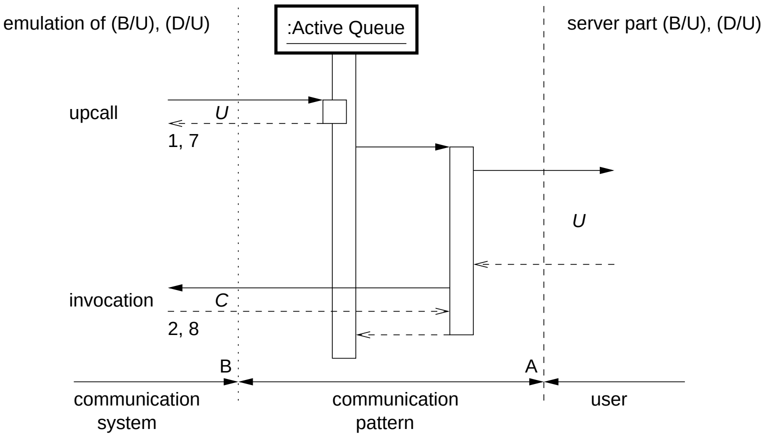


Figure 5.68: *An active queue inside the server part emulation of the (B/U) and (D/U) interaction patterns.*

An obvious but unsuited processing model for use inside the server part of the (B/U) and the (D/U) interaction pattern is the *active queue* shown in figure 5.68. The upcalls ① and ⑦ are solely used to enqueue the requests and thus behave like ③/(O) respectively ⑦/(I2). The thread of the active queue calls the emulated synchronous user interface (U). As soon as that upcall returns, the active queue calls the client parts of ② and ⑧ before the next request is processed. Since there is no chance to circumvent the pattern internal processing model at the user level, one cannot circumvent the processing order enforced by the active queue. The buffer of the active queue decouples the upcall but does not allow a user level threading model to override the processing order introduced by the queue due to the server side synchronous pattern initiation mode (U).

Blocking the server side upcall of the (C/U) interaction ⑤ is acceptable since the upcall solely stores the request in a buffer and since the resources needed to access the buffer do not depend on the upcalling thread. The same holds true for the interactions ②, ④, ⑧ and ⑩ that forward responses to the buffers and that signal the arrival of the response. However, the upcalling thread of ⑤ gets blocked as soon as there is not enough space left in the server side buffer of the (C/W) interaction pattern to accept the request. Thus, one has to make sure that the blockade does not affect any shared resources like system level buffers and threads. The responsibilities of the upcalling thread should not comprise other vital tasks of a component like the communication system, for example. Furthermore, one should not share such communication resources where filled up buffers prevent other interactions from making progress. It is important to note that the upcalls of ②, ④, ⑧ and ⑩ are not affected since they are always able to get rid of the response message. The difference to the (C/W) interaction pattern is that with a successful invocation at the client side of the interaction pattern, one also has successfully set up the infrastructure to accept a response. Even if the response itself is too large, one can discard it but one can still use the prepared infrastructure to inform the client that the response got discarded. The upcall of the (C/U) interaction pattern thus never has to wait until enough memory is available. Therefore, even if the delivery gets delayed, a congestion always gets resolved and the serialization of all upcalls belonging to the (C/U) interaction patterns ②, ④, ⑧ and ⑩ is possible. That is the reason why these upcalls can share a thread to perform all the upcalls and why shared buffers and communication resources are allowed for those (C/U) interaction patterns.

In principle, the (C/U) interaction pattern gives full control over the used resources. Due to the one-way communication, it allows to implement any kind of processing model above the communication system level without wasting upcalling threads. Threading models can be implemented either inside the emulation of the interaction patterns or at the user level. The latter is done by means of different types of handlers. The user level processing models can be finetuned without being restricted to given threading models of the communication system. The only patterns that depend on a threading model beneath the user level, either inside the emulation or even at the level of the communication system, are the server parts of the (B/U) and the (D/U) interaction patterns. Nevertheless, threading models of the communication system can still be exploited. Due to the *delivered* policy of the (C/U) interaction pattern, the client part invocation method is decoupled from the server part upcall and it returns independently of the completion of the server part upcall. Thus, no additional arrangements have to be taken to achieve the required level of decoupling.

5.6.5.3 Nested Calls of Interaction Patterns

Interaction patterns calling each other must be aware of resources they share to avoid deadlocks. At the level of the emulation of the interaction patterns, the upcall of the server part of the (C/U) interaction pattern invokes the client part of another (C/U) interaction pattern. At the level of the communication patterns, a server part handler might invoke a client part method with a *processed* semantics and thus gets blocked until the response arrives. The upcalling thread of the first interaction pattern that now blocks in the second interaction pattern might just be needed to keep the communication alive that can provide the expected response. Thus, the safe way to invoke a blocking method of an interaction pattern is to always use a thread that is decoupled from the internals of an interaction pattern. The decoupling can be provided either by a user level processing model or by a threading model inside the interaction pattern. In principle, separate threads can be avoided if the resources that are required by the upcalling thread are available independently of the upcalling thread. That is, separate threads can be avoided as long as the availability of required resources does not depend on releasing the upcalling thread.

5.6.5.4 The Decoupling Characteristic of the Delivered Policy

The *delivered* policy of the (C/U) interaction pattern is mandatory for the emulation of the other interaction patterns. Of course, the processing models of the upcalls of the split protocol are directly related to the ones shown in figure 5.60. Thus, a client part asynchronous invocation depends on the *delivered* policy of the (C/U) interaction pattern to always return at ①. Figure 5.69 and figure 5.70 show the emulation of the (D/V) interaction pattern by means of two independent (C/U) interaction patterns. Due to the *delivered* policy, the client part asynchronous characteristic (D) is not affected by the server side processing model and the processing model affects the server reactivity only. The *delivered* policy ensures that the client part invocation always returns after ① and independently of the processing models used at the server side. Of course, the upcall is blocked for the time required to perform the processing or rather to forward the request to the active handler.

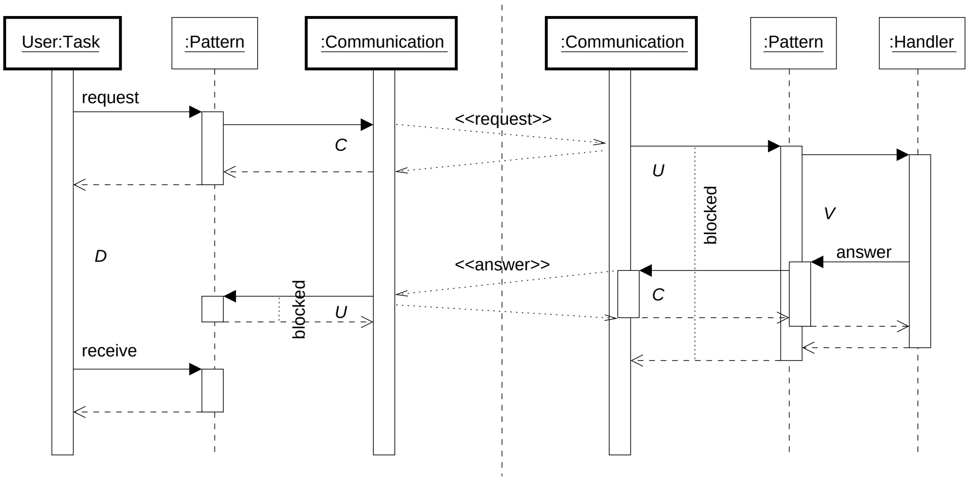


Figure 5.69: *A passive handler and a (D/V) interaction pattern emulated by two (C/U) interaction patterns.*

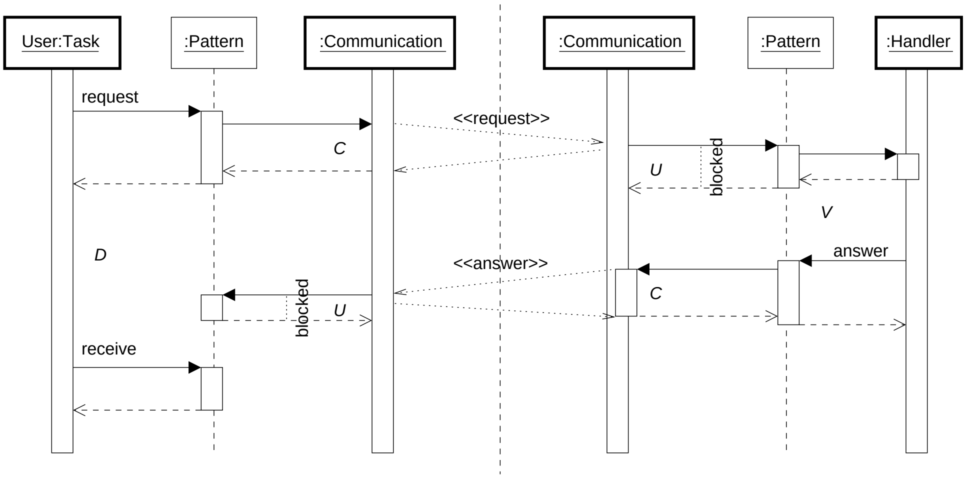


Figure 5.70: *An active handler and a (D/V) interaction pattern emulated by two (C/U) interaction patterns.*

Figure 5.71 shows the same situation but with a synchronous interaction that implements a *processed* policy. It uses a synchronous one-way (B/U) interaction pattern instead of the asynchronous (C/U) interaction pattern. Due to the active handler, the client side characteristic is still matched.

However, the situation changes completely if a passive handler is used as shown in figure 5.72. Now, the client side invocation returns only after the server side processing is completed and thus the *delivered* semantics to be emulated is violated. The resulting tight coupling of the client and the server is solely caused by the *processed* policy of the synchronous interaction model. Returning from the client side request method is tantamount to an already available answer. The interaction still works correctly if steps are taken to avoid deadlocks but a client side asynchronous characteristic does not show the expected decoupling anymore and thus becomes obsolete. Furthermore, without support from the communication system, a client side interface method could not be aborted before the server side processing is completed.

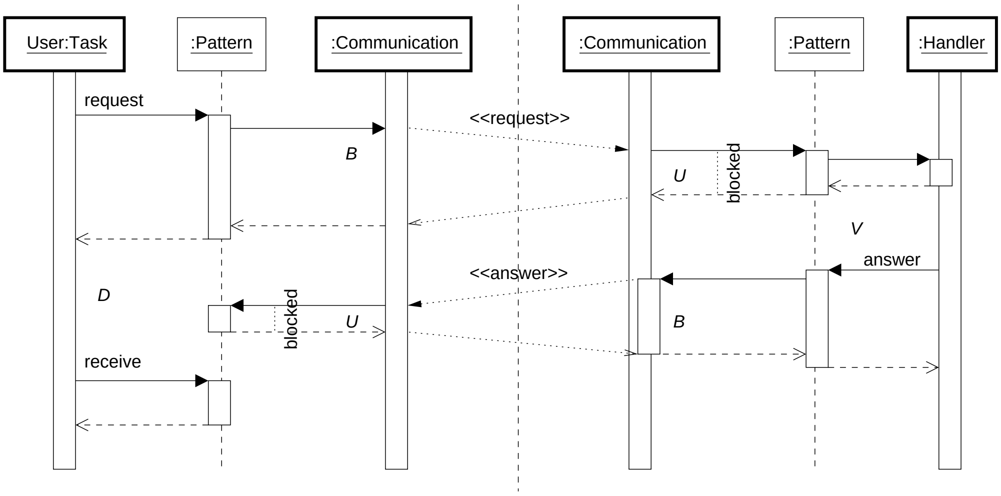


Figure 5.71: *An active handler and a (D/V) interaction pattern emulated by two (B/U) interaction patterns.*

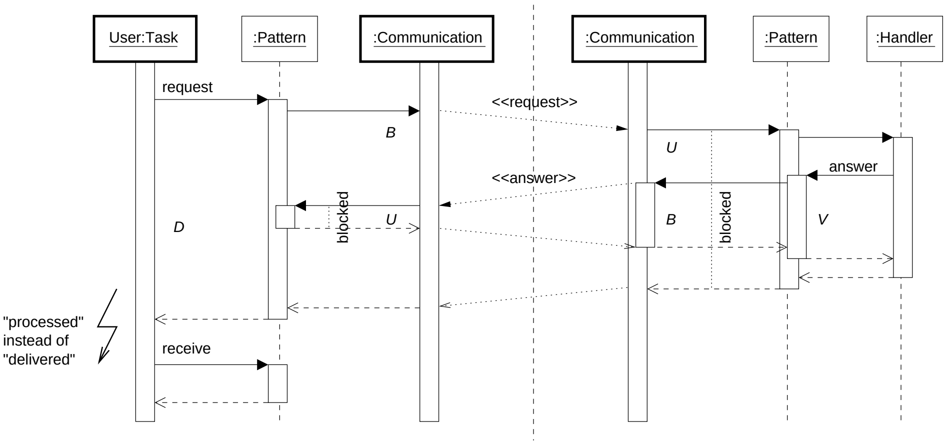


Figure 5.72: *A passive handler and a (D/V) interaction pattern emulated by two (B/U) interaction patterns.*

Potential sources of deadlocks that arise solely due to the fact that a protocol is put on top of a synchronous communication become obvious with the interaction illustrated in figure 5.72. If the request holds a lock that is needed to process the received answer, then this results in a selflock since the lock cannot be acquired again. Even recursive locks do not prevent from selflocks since the thread to process the incoming answer is in most cases different from the thread holding the lock in the *request* method. As can be seen in figure 5.69, the asynchronous implementation of the same interaction causes no trouble.

The challenge is to return to the server *before* the processing is completed that is at the latest at ⑦ of (I2) in figure 5.67. According to figure 5.60, one has to leave the upcall as soon as possible as marked by ③ which requires an active processing model. The processing models (M) and (N) cause the client part invocation to return only after the server part processing is completed as labeled by ② respectively ⑦/(I1) in figure 5.67. Since the client part characteristics of the interaction patterns have to be independent of the user level processing models, one has to provide an active processing model on top of the synchronous communication to achieve the asynchronous characteristic of the (C/U) interaction pattern that forms the interface to the communication patterns. Basically, that is the same situation as with the emulation of the synchronous server characteristic (U) described in the previous section. Instead of returning to the server as soon as possible to avoid a blockade of the upcall, one now has to return from the upcall as soon as possible to decouple the client side invocation from the server side processing.

With a *processed* policy, the client part user interface method of the (C/U) interaction pattern and the client part request method of both the (D/U) and the (D/V) interaction patterns return only after the server part upcall was completed. Thus, the client side asynchronous characteristic becomes obsolete without decoupling the server side upcall from the request processing. However, the desired asynchronous characteristic is achieved only with threading models that are located above the communication system level. Threading models of the communication system cannot be exploited since the client side return is directly related to the return of the server part upcall and in case of exploiting threading models of the communication system, one can return from the upcalling thread only after that thread had been used to perform the processing. The global characteristics of the client part interface of the (B/U) and the (B/V) interaction patterns are not affected by a *processed* policy. However, their client part interface methods remain in the interaction ① or ③ until the interaction with the server is completed. Depending on the server side processing model, one has to await the completion of the user level processing. Thus, the client part user interface can be aborted either only after the server side processing is completed or if appropriate support from the communication system is available to abort blocking calls into the communication system.

In all cases, two-way interaction patterns require means to prevent from deadlocks at the client part since the answers are already provided while the request is not yet finished. In table 5.43, the upcalls ②, ④, ⑧ and ⑩ must only access resources that are not locked by the corresponding interaction ①, ③, ⑦ and ⑨. The upcalls ②, ④, ⑧ and ⑩ solely invoke the client part emulation of the interaction patterns and cannot get blocked by a user level implementation. The emulation of the interaction patterns can be implemented such that the processing inside the emulation does not suffer from potential deadlocks due to a *processed* policy. Signalling the availability of a response does never block but the upcalls need to acquire a mutex that protects the buffer that stores the answer. The emulation is not affected by a *processed* policy in case that this mutex is not locked while the interactions ①, ③, ⑦ and ⑨ are invoked.

To summarize, using a *processed* policy instead of the *delivered* policy introduces a tight coupling between the client and the server part of the emulated interaction patterns. The *delivered* policy is required with the interactions ①, ③, ⑦ and ⑨. The other interactions including the interaction ⑤ of the (C/W) interaction pattern are not affected since their upcalls are not involved in the actual processing of a request. They either provide a response or store a request in a buffer which does not require the decoupling of the *delivered* policy. Compared to a *delivered* policy where the invocations of ②, ④, ⑤, ⑧ and ⑩ return before the upcall is invoked, awaiting the completion of the upcall in case of a *processed* policy is completely acceptable for those interactions.

5.6.5.5 The Implicit Acknowledgment of the Delivered Policy

A mandatory characteristic of the *delivered* policy of the (C/U) interaction pattern is the implicit acknowledgment. It gives the sender the feedback that the message is going to be processed. In contrast, as illustrated in figure 5.73, a *reliable send* policy gives no feedback if a message is dropped due to a nonexisting recipient. Even if no message gets lost, one basically never knows whether the recipient still exists and whether one can expect a response message. Thus, one has to take further arrangements to either achieve the *delivered* policy in case it is not available or to extend the protocol above the interaction patterns such that one can also do with the weaker *reliable send* policy.

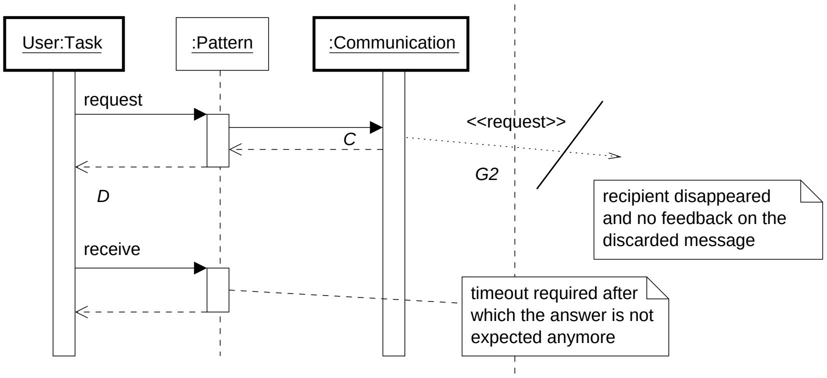

User:Task
:Pattern
:Communication
request
<<request>>
C
G2
D
receive
recipient disappeared and no feedback on the discarded message
timeout required after which the answer is not expected anymore

Figure 5.73: *A* reliable send *policy. No feedback is given on the discarded message if the recipient disappeared.*

5.6.5.6 Converting a Reliable Send Policy into a Delivered Policy

The first option is to emulate the *delivered* policy by introducing additional acknowledgment messages. These are handled on top of the *reliable send* policy of the communication system and beneath the (C/U) interaction patterns that are visible at the interface to the communication patterns. However, this is not as easy as it appears at first glance. Since no message gets lost, the acknowledgment fails to appear only if the recipient is not available anymore. Getting an acknowledgment provides the information that the message was delivered successfully. In contrast thereto, an absent acknowledgment gives no further information on the state of the message. It could either got discarded due to a disappeared recipient, can still be on its way towards the recipient or got delivered but the acknowledgment is still on its way back towards the sender. In principle, acknowledgment messages drastically increase the overall traffic.

The proper solution is to generate the acknowledgment at the level of the communication system. However, this requires support from the communication system since the acknowledgment messages have to be generated automatically as soon as a message is delivered or is dropped. Then, one always gets a feedback on the whereabout of a message. Generating the acknowledgment messages without support from the communication system is tricky since one has to detect that a recipient is not available anymore. Even redirecting all messages that address no regular recipient towards a component that can at least generate acknowledgment messages containing the appropriate status codes is not easily implemented on top of a communication system.

If messages without a regular recipient get dropped silently by the communication system, one has no chance to generate the appropriate acknowledgment messages at a level above the communication system. If one cannot be sure that the recipient still exists, one has to use timeouts when awaiting the acknowledgment. An absent acknowledgment can now also mean that the message has been delivered but the sent back acknowledgment returned too late. Then, the message is processed at the recipient even if the sender experienced a timeout and the acknowledgment arrives anyhow. Thus, in case of a timeout, assumptions on the state of the recipient can get out of sync which is in contrast to a *delivered* policy. Furthermore, one has to take special care to prevent the interaction from getting messed up by outdated messages. A general problem with timeouts is to properly define the maximum waiting time. A too large value results in a reduced efficiency caused by far too long waiting times. With a too small timeout value, still existing recipients can get not accessible anymore.

The second option is to handle disappeared recipients at the level of the protocol used above the interaction patterns that is inside the communication patterns. This shift allows to map the (C/U) interaction patterns onto the weaker *reliable send* policy without requiring further glue logic. However, that is possible only if the interactions of both parts of a communication pattern are protected by an appropriate protocol as it is the case for the connection oriented protocol presented in section 5.6.6.

5.6.5.7 Converting a Processed Policy into a Delivered Policy

Implementing the (C/U) interaction pattern on top of an interaction model with a *processed* policy requires to circumvent the tight coupling introduced by the *processed* policy. The affected upcalls are ①, ③, ⑦ and ⑨. A server part processing model has to decouple the communication system upcall from the emulated interaction pattern.

As already pointed out, the decoupling is achieved only if the server part processing models of the (C/U), (B/U), (B/V), (D/U) and (D/V) interaction patterns do not use threading models of the communication system. These are unsuited since in case of a *processed* policy, the client side invocation returns only after the server part thread is released. Thus, in that case, threading models of the communication system are still useful to decouple concurrent requests such that these are not already serialized before they get forwarded to the actual processing models. That becomes particularly important in case of a congestion. In general, each threading model beneath the user level results in wasted resources as soon as a user level threading model is put on top. With the interaction patterns (C/U), (B/V) and (D/V), a user level threading model is sufficient to allow the upcall to return immediately to achieve the *delivered* policy. Since the upcalls are one-way interactions only, active user level processing models do not block the upcalling thread besides the time needed to complete the forward. Of course, it depends on the capacity of the user level processing model whether the upcall still gets blocked until either enough buffer space or further threads are available. With the (B/U) and the (D/U) interaction patterns, the decoupling could also be performed by the threading model that is anyway needed inside the emulation.

The server side decoupling mechanism can be inserted at different places as illustrated in figure 5.74. Depending on the layer the decoupling is performed, different parts of a component get coupled and are thus affected by a bottleneck in one of the processing units. In case of (A), every single interaction pattern gets decoupled separately. We first assume that individual upcalls from the communication system do not share any resources and that every upcall is performed by a separate thread. Thus, blocking the resources beneath one upcall does not affect any other upcalls. However, that model results in far too many threads and the upcalling thread of the communication system is

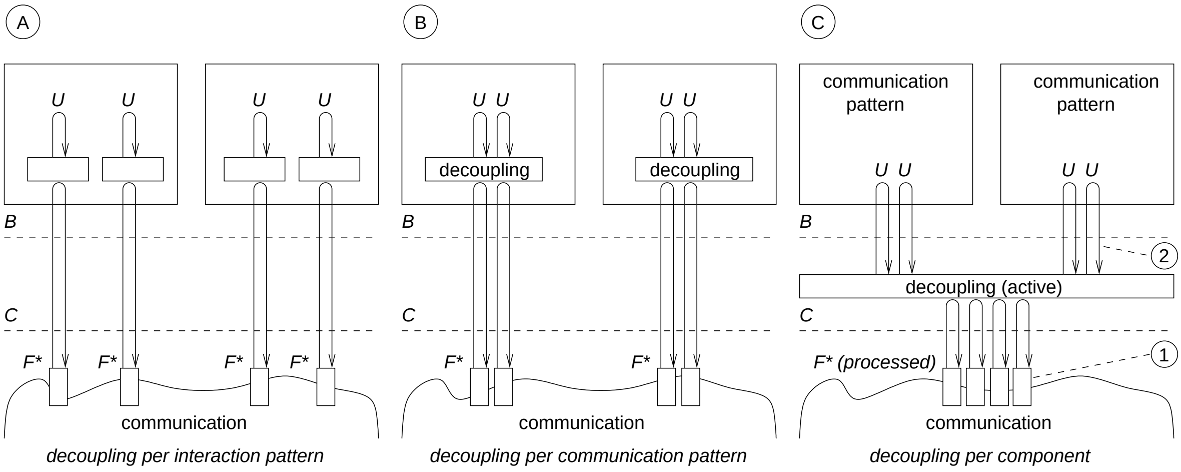


Figure 5.74: *A decoupling mechanism to convert the* processed *policy into a* delivered *policy.*

even wasted since it is solely used to pass on requests to threads that perform the actual processing. Since all interaction patterns of a communication pattern are processed independently of each other, one has to take additional measures to ensure that certain administrative interactions, that are based on different interaction patterns, get processed in the desired order. The number of threads can be cut down by performing all upcalls of the communication system by a single thread. Then, however, only such upcalls are allowed to get blocked that do not at all depend on any communication activities. All other upcalls need to release that thread so that the communication keeps alive. For these upcalls, one needs either appropriate buffers that always accept the requests or enough threads that directly accept the request. In case that the number of threads or that the buffer capacity is limited, one runs the risk of a deadlock. In case of (B), the interaction patterns of one communication pattern share a decoupling mechanism and in case of (C), the decoupling mechanism is even shared by all communication patterns of a component. Of course, sharing resources can be very efficient but only at the price of introducing further dependencies that have to be evaluated and checked carefully.

At first glance, buffers in various forms like active queues, for example, appear to be a suitable mean to decouple the upcalling thread. The communication system upcalls are used to enqueue the requests and thus return immediately in compliance with the desired *delivered* policy. Further threading models then operate on the buffer and assign entries to threads in accordance with different policies. However, a general problem with buffers is to properly reject new entries in case of a shutdown of an interaction. Since the split protocol is based on one-way interactions, there is no back channel to report a rejected request. Once the upcall ① is invoked, one has to enqueue the request and once enqueued, the request has to be processed. Otherwise, the *processed* policy is converted into an *unreliable send* policy only and not into a *delivered* policy. In case entries should not be accepted anymore, one must already reject a request at the level of the communication system by closing the upcall ① of the concerned interaction. Addressing the not anymore available upcall then provokes an appropriate error at the level of the communication system and indicates that the delivery of the request was not successfully accomplished. The already enqueued entries can now still be processed by the upcalls ②. Buffers thus require means to selectively deactivate the upcalls ① without influencing upcalls of other interaction patterns that share the buffer. As soon as the buffer is placed outside a communication pattern, one needs elaborate interactions via appropriate external interfaces to detect whether there are still to be processed entries available. Unfortunately, these modify the generic interface between the communication patterns and the communication system and are thus not acceptable.

5.6.5.8 Mapping of the (C/U) Interaction Pattern

The (C/U) interaction pattern maps straightly onto the interaction model (G1). That interaction model already provides an asynchronous interaction and thus the client side invocation always returns after the request is delivered as marked by ① in figure 5.67. Furthermore, server side processing models affect the reactivity of the server only and do not influence the client part characteristics. The same holds true for the interaction model (H) in case the back channel can be ignored. Ignoring the back channel means that no client side invocation is needed to close an interaction that is otherwise kept open.

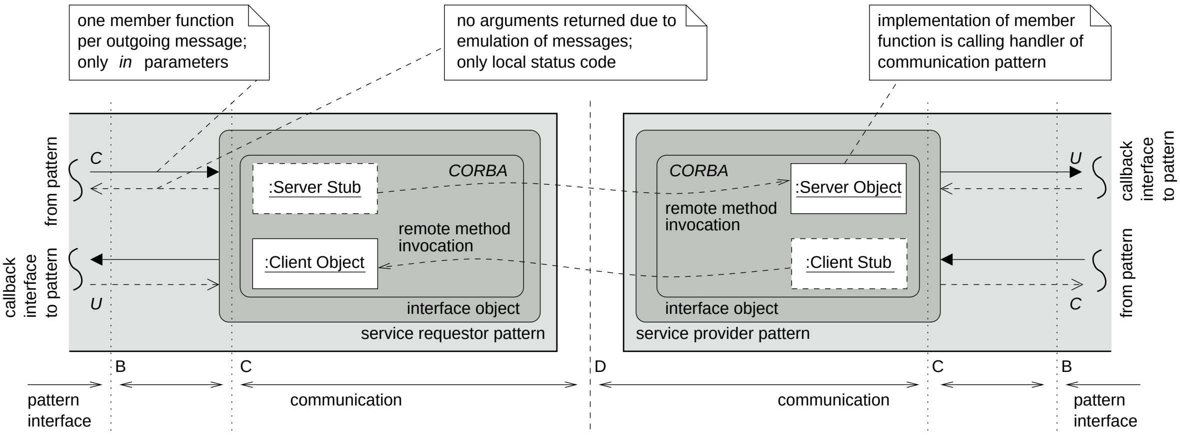

one member function per outgoing message; only *in* parameters
no arguments returned due to emulation of messages; only local status code
implementation of member function is calling handler of communication pattern
from pattern
callback interface to pattern
C
U
CORBA
:Server Stub
remote method invocation
:Client Object
interface object
service requestor pattern
CORBA
:Server Object
remote method invocation
:Client Stub
interface object
service provider pattern
U
C
callback interface to pattern
from pattern
B
C
D
C
B
pattern interface
communication
communication
pattern interface

Figure 5.75: The basics of the communication on top of CORBA.

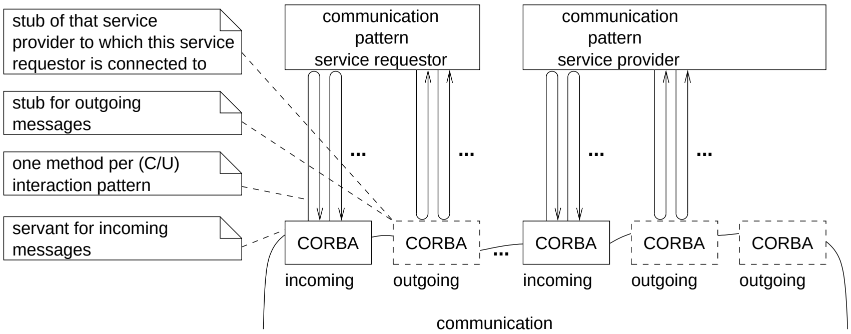

stub of that service provider to which this service requestor is connected to
stub for outgoing messages
one method per (C/U) interaction pattern
servant for incoming messages
communication pattern service requestor
communication pattern service provider
...
...
...
...
CORBA
CORBA
...
CORBA
CORBA
CORBA
incoming
outgoing
incoming
outgoing
outgoing
communication

Figure 5.76: *Mapping the (C/U) interaction pattern onto the* CORBA oneway *model respectively the* CORBA AMI.

With *CORBA*, one can map the (C/U) interaction patterns onto methods as shown in figures 5.75 and 5.76. Several methods can be pooled into a single object as long as the corresponding interaction patterns can always be activated or deactivated jointly and never need to be activated or deactivated individually. The worst case requires one object per (C/U) interaction pattern. The methods have to be declared as *oneway* with a *sync with server* policy to achieve the interaction model (G1). *CORBA* then ensures that in case of deleting the servant object, already received and acknowledged requests still get processed and do not get lost but further requests are already rejected. Thus, no additional precautions have to be taken to prevent the *delivered* policy from getting violated when deleting the recipient of a request. The *CORBA* object life cycle management already handles the hidden buffers in compliance with the *delivered* policy.

Another option with *CORBA* is to use the *AMI* according to the interaction model (H) and to simply ignore the back channel. As already outlined, the back channel of the *AMI* cannot be used to replace the second (C/U) interaction since one then cannot circumvent the restrictions of the synchronous server side interface with respect to user level processing models.

Since the split protocol handles blocking calls outside the communication system and since the interaction patterns stay in the *CORBA* communication system for a very short time only due to the *delivered* policy, no support from the communication system is needed to abort client side blocking calls.

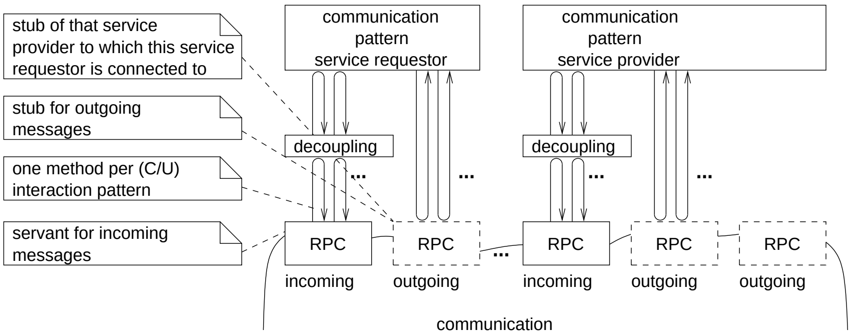

stub of that service provider to which this service requestor is connected to
stub for outgoing messages
one method per (C/U) interaction pattern
servant for incoming messages
communication pattern service requestor
communication pattern service provider
decoupling
decoupling
...
RPC
RPC
RPC
RPC
RPC
incoming
outgoing
incoming
outgoing
outgoing
communication

***Figure 5.77:** The (C/U) interaction pattern on top of a synchronous communication like the* RPC *respectively standard* CORBA *interactions.*

Mapping the (C/U) interaction pattern onto the interaction model (F) requires a decoupling mechanism. With both the standard *CORBA* synchronous interaction and the standard synchronous *remote procedure calls (RPC)*, the upcalls of the servant object have to be routed via a decoupling mechanism as shown in figure 5.77 and everything outlined on how to convert a *processed* policy into a *delivered* policy has to be considered. As already explained, the decoupling mechanism can be placed at various places but always above the servant object. Again, one has to take care that only those interaction patterns share a servant object that can be activated or deactivated jointly.

Basically, a shared decoupling mechanism merges requests that are already separated by the *one-method-per-message* mapping strategy. A shared decoupling mechanism abandons the advantages of the *CORBA* and the *RPC* mapping. Since one cannot circumvent the message dispatching mechanism, one could also do with one servant per component that accepts all incoming messages and a single upcall that forwards them for further decoding and dispatching. Then, however, the successful delivery of a message can only be interpreted according to a (G2) interaction model since one has already accepted the upcall before one can detect that the recipient does not exist anymore.

The interaction model (F) provides a back channel that is not used with the mapping of the (C/U) interaction pattern. The reason is the need for circumventing the synchronous server part interface. Nevertheless, the back channel could be exploited to provide the acknowledgment whether the request got accepted by the decoupling mechanism. If multiple upcalls are grouped into a single servant object, one could selectively reject upcalls at the level of the decoupling mechanism even if the shared servant object still existed and thus performed the upcall. The returned status code indicating a rejected upcall has to be treated in the same way as an error from the communication system indicating that the recipient is not accessible. However, this procedure is applicable only on top of the interaction model (F) where an unused back channel is available. Furthermore, the required modifications on the interface of the communication patterns to the communication system are not generic with respect to other interaction models like the interaction model (E).

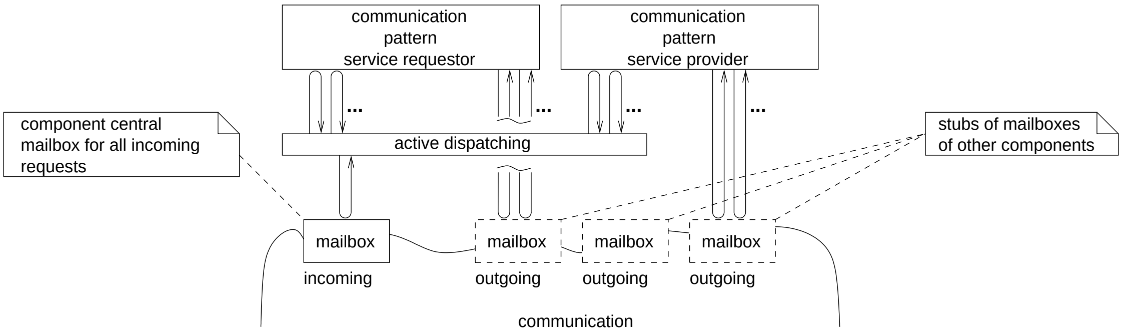

communication pattern service requestor
communication pattern service provider
component central mailbox for all incoming requests
active dispatching
stubs of mailboxes of other components
mailbox
incoming
mailbox
outgoing
mailbox
outgoing
mailbox
outgoing
communication

Figure 5.78: *The (C/U) interaction pattern on top of* mailboxes *respectively* TCP sockets.

Mapping the (C/U) interaction pattern onto the interaction model (E) requires a mechanism to convert the characteristic (W) into the characteristic (U). A threading model on top of the mailbox performs the polling and invokes the appropriate upcalls. A *mailbox* based messaging system behaves exactly like a *FIFO* queue and the polling thread converts the mailbox into an active queue. Again, a mailbox can be shared by several interaction patterns or even by several communication patterns and it depends on the placement of the mailbox whether elaborate dispatching mechanisms are needed. Figure 5.78 illustrates a component central mailbox. Normally, one cannot selectively reject specific messages from being accepted respectively from being entered into the mailbox. Typically, one has to shutdown the mailbox to prevent it from accepting further entries. Thus, a mailbox can be shared only for interaction patterns that can all be shutdown at the same time. A shared mailbox results in a (G2) interaction model if one only detects that the recipient does not exist anymore after already having accepted a message.

An important aspect of mailboxes is that all messages are serialized. That becomes a severe problem as soon as a message of a shared mailbox cannot be forwarded to the corresponding interaction pattern. Due to the serialization, this also prevents the messages that are enqueued behind from getting through. As already explained in the context of the shared decoupling mechanisms that are illustrated in figure 5.74, blocking of shared resources is allowed for those upcalls only that do not at all depend on further communication activities. For all other upcalls, one either needs enough threads or enough buffer capacity above the shared mailbox such that one always is able to empty the shared mailbox.

In principle, *TCP sockets* behave in the same way as mailboxes since a shared TCP socket also accepts the incoming data before the dispatching mechanism detects that the recipient rejects the request or is even not available anymore. As well as one mailbox per interaction pattern is not feasible due to the required resources, one TCP socket per interaction pattern also wastes far too many resources and is unfeasible, too. A shared socket again results in a (G2) interaction model if messages have to be discarded due to a disappeared recipient after they have been accepted by the socket.

Mapping the (C/U) interaction pattern onto the interaction model (G2) requires substantial efforts to achieve the *delivered* policy. Thus, even that the *split protocol* requires one-way interactions only, the interaction patterns can still not easily be mapped onto many message based systems. Even

CORBA oneway methods cannot be used without substantial extensions if the *sync with server* policy is not available. However, the interaction model (G2) cannot be ignored since it covers a large family of communication systems and depending on the kind of implementation, many other interaction models also result in a (G2) interaction model.

5.6.5.9 Summary on the Split Protocol

An important demand on the communication patterns is to decouple a service requestor from its service provider. That includes the asynchronous operation of both parts. In principle, server part processing models are not allowed to influence the characteristics of a client part user interface. Furthermore, the communication patterns have to be implementable on top of many different communication systems and the migration between communication systems should not require substantial modifications inside the communication patterns.

As shown, the naive usage of synchronous interactions can introduce strong couplings and can even make asynchronous user interfaces obsolete. Inexpertly chosen structures not only require extensive resources but above all these are wasted with some of the needed user level processing models. Although the *split protocol* can be mapped onto many different communication systems due to its one-way communication, the requirements with respect to decoupling the communicating parts in combination with the required guarantees of an interaction cannot easily be achieved on all types of communication systems. Even a buffer can already convert a *delivered* policy into a *reliable send* policy or even into an *unreliable send* policy.

The *split protocol* provides an uniform interface between the communication system and the communication patterns and requires one-way interactions only. Additionally, except for the (B/U) and the (D/U) interaction patterns, user level processing models can be implemented easily. Indeed, the *split protocol* remains demanding with respect to the *delivered* policy. However, as described in the following section, the connection oriented design of the communication patterns circumvents those difficulties.

5.6.6 The Communication Protocol of the Communication Patterns

This section presents the *connection oriented split protocol* that is used inside the communication patterns. It is based on the *split protocol* and thus also requires one-way interactions only, possesses the same lean interface between the communication patterns and the communication system and provides the same flexibility with respect to user level threading models. The major extension is the full exploitation of the connection oriented design of the communication patterns. Besides the interactions related to the connection management, all interactions between a service requestor and a service provider part of a communication pattern are monitored by the connection management of the communication pattern. Changes to a connection can be made at any time and the connection management assumes the responsibility for the proper handling of affected interaction patterns. Once a service requestor is connected to a service provider, both inform each other about getting unreachable. Communication patterns always know when their opponent disappears and thus also know which messages reach their destination and which messages can be awaited. The consequence is that the requirements on the interaction pattern that is used as interface between the communication patterns and the communication system can be further alleviated. In principle, the *connection oriented split protocol* achieves the features of the *delivered* policy at the level of the communication patterns and relieves the interface to the communication system from that task.

The big step is that a (C⋆/U) interaction pattern with a *reliable send* policy instead of a (C/U) interaction pattern with a *delivered* policy is now sufficient as interface to the underlying communication system. That makes it much easier to map the generic interface onto interaction models of widespread communication systems. The uniform (C⋆/U) interface of the *connection oriented split protocol* can even be mapped onto the (G2) interaction model. One does neither require any support from the communication mechanism to automatically generate acknowledgment messages nor does one run into the pitfalls of timeout procedures. Since a *reliable send* policy is now sufficient, decoupling mechanisms and buffers that convert a *processed* policy into a *reliable send* policy, are not critical anymore. Above all, most of the administrative interactions do not depend on the decoupling properties of the (C⋆/U) interaction pattern. Due to the presented locking strategies, they even work on top of a *processed* policy without requiring any further decoupling. That significantly simplifies the implementation of the *connection oriented split protocol* since the decoupling has to take effect mainly for the service related interactions and can thus be done by the user level processing models of the handlers.

5.6.6.1 The Interaction Patterns Required by the Communication Patterns

The interaction patterns that are required by the communication patterns due to their currently assigned access modes are summarized in table 5.44. The table lists only those interaction patterns that are directly related to the user interface of a communication pattern. All communication patterns require additional (B/U) interaction patterns to perform a *connect* and a *disconnect* and a (C/U) interaction pattern to inform connected service requestors about the destruction of a service provider. The *query* pattern requires an additional (C/U) interaction to inform the service provider about a discarded request. The *push* patterns require additional (B/U) interaction patterns to *subscribe* and to *unsubscribe* from service providers and the *event* pattern to *activate* and to *deactivate* events. The administrative interactions are fully processed inside the communication patterns without any user interaction or user provided handlers. Thus, those interaction patterns need not to be decoupled from user influences. The *wiring* pattern is internally based on the *query* communication pattern. Since it is not based on the basic interaction patterns, it is thus not listed in table 5.44.

communication pattern	send	query	push	event
required interaction pattern	C/U	B/V D/V	C/W	C/U C/W
client part of interaction pattern located at	service requestor (client)	service requestor (client)	service provider (server)	service provider (server)
server part of interaction pattern located at	service provider (server)	service provider (server)	service requestor (client)	service requestor (client)

Table 5.44: *The interaction patterns required by the communication patterns due to their current access modes.*

The *send* pattern provides a one-way communication from the service requestor to the service provider with a *delivered* policy that decouples the server side processing from the client side invocation. The access mode of the service requestor conforms to the asynchronous one-way characteristic (C) and the access mode of the service provider to the synchronous pattern characteristic (U). The *send* pattern thus requires the (C/U) interaction pattern with the client part at the service requestor. The server part (U) executes the user provided handler for commands. The emulation of the (C/U) interaction pattern requires a single ($C^\star$/U) interaction pattern.

The *query* pattern provides a two-way communication from the service requestor to the service provider with synchronous and asynchronous access modes. The access mode of the service provider conforms to the asynchronous pattern characteristic (V). The *query* pattern thus requires the (B/V) and the (D/V) interaction patterns with the client parts at the service requestor. Each request receives one answer at maximum and thus the maximum size of the buffer at the client part of the (D/V) interaction pattern corresponds to the number of simultaneously open requests. The server part (V) executes the user provided handler for queries. The client part characteristic (B) of the (B/V) interaction pattern is emulated on top of the client part characteristic (D) of the (D/V) interaction pattern. The emulation of the (D/V) interaction pattern requires two ($C^\star$/U) interaction patterns.

The *push* patterns provide a one-way communication from the service provider to the service requestor with a *delivered* policy. Based on a user invocation mode, one can get the latest update or await the next one without being forced to react to every incoming update. The *push* patterns thus require the (C/W) interaction pattern with the client part located at the service provider. Since updates overwrite each other and since only the latest update is stored, a buffer size of one is sufficient at the server part of the (C/W) interaction pattern. The emulation of the (C/W) interaction pattern requires one ($C^\star$/U) interaction pattern.

The *event* pattern provides a one-way communication from the service provider to the service requestor with a *delivered* policy and both a user invoked and a handler based access mode at the service requestor. The *event* pattern thus requires the (C/U) and the (C/W) interaction patterns with the client parts located at the service provider. Again, the emulation of both interaction patterns share the same ($C^\star$/U) interaction pattern. Since an event fires only once or overwrites outdated firings in case it fires multiple times, the server part buffer size of the (C/W) interaction pattern corresponds to the number of simultaneously activated events. An optionally provided handler for firing events is executed by the server part (U). If a handler is registered at the service requestor, only the (C/U) interaction pattern is active and if no handler is registered, only the (C/W) interaction pattern is active.

5.6.6.2 The ($C^\star$/U) Interactions of the Communication Patterns

The interactions between both parts of a communication pattern are emulated on top of the ($C^\star$/U) interaction pattern. These interactions are listed in table 5.45 and 5.46. Each line corresponds to one

(C⋆/U) interaction. The format is described by means of the *CORBA IDL*. The interaction patterns that are required besides the administrative interactions are marked by D and correspond to the interaction patterns listed in table 5.44. Administrative interactions are marked by a star and corresponding request/response pairs of administrative interactions are denoted by Rx and Ax, respectively. Figure 5.79 illustrates the (C⋆/U) interaction patterns of the *send* communication pattern.

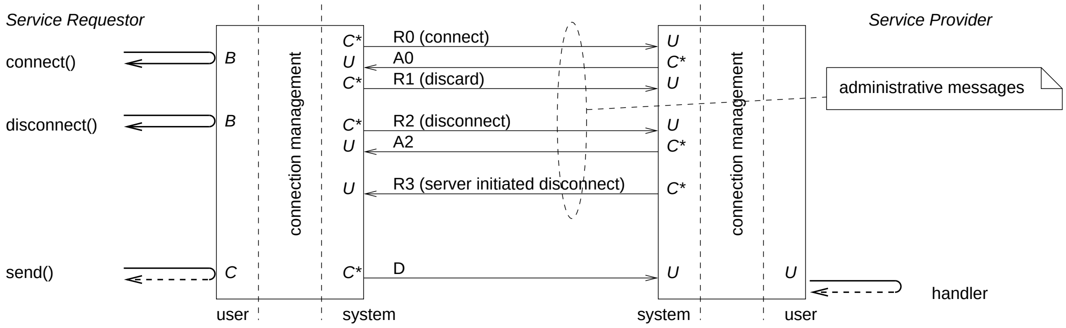


Figure 5.79: *The (C⋆/U) interactions of the* send *communication pattern.*

Before one can use a service requestor, it needs to be connected to a service provider. The connection management consists of the messages R0, A0 and R1 for the *connect* procedure and the messages R2 and A2 for the *disconnect* procedure. The *server initiated disconnect* message R3 is needed to properly disconnect service requestors in case that a service provider gets destroyed. The messages R0, R1 and R2 can be emitted by all service requestors. The acknowledgment messages A0 and A2 and the message R3 can be emitted by all service providers.

The *connect* message provides the address of the service requestor and a connection identifier to the service provider. The address enables the service provider to inform the service requestors that are connected to it in case it gets destroyed. The connection identifier is generated by the service requestor and uniquely identifies each connect procedure. It is returned by the message A0 so that one can identify outdated acknowledgments that are possible due to the underlying *reliable send* policy. The status simply indicates whether the connect has been accepted or not. A *connect* cannot be performed if the service provider is either not yet ready or is already in the process of destruction. In both cases, service providers do not accept any connections from service requestors. The role of the *discard* message and the details of the *connect* procedure are explained in section 5.6.6.10.

The address of the service requestor is needed with a *disconnect* to remove the appropriate entry from the list of connected service requestors. Once the service requestor received the acknowledgment, it knows that from now on no further messages that are related to the just closed connection can be on their way towards the service requestor. The disconnect procedure does not need an identifier since there can be no outdated acknowledgments as it is the case with the connect procedure. The details are again illustrated in section 5.6.6.10.

The *server initiated disconnect* is invoked if the service provider wants to remove all its service requestors. That is needed if the service provider gets destroyed. The message R3 contains the connection identifier that enables the service requestor to check whether the order to get disconnected is still relevant. The connection identifier prevents a meanwhile newly established connection to another service provider from getting closed in case that a *server initiated disconnect* at the service provider coincides with a *disconnect* at the service requestor that is immediately followed by a *connect* to another service provider. The details are also explained in section 5.6.6.10.

Pattern	Receivable Messages
Send	SmartSendClientPattern (Service Requestor):
⋆	*A0 oneway void acknowledgmentConnect(in long cid,in long status)*
⋆	*A2 oneway void acknowledgmentDisconnect()*
⋆	*R3 oneway void serverInitiatedDisconnect(in long cid)*
	SmartSendServerPattern (Service Provider):
⋆	*R0 oneway void connect(in SmartSendClientPattern address,in long cid)*
⋆	*R1 oneway void discard(in SmartSendClientPattern address)*
⋆	*R2 oneway void disconnect(in SmartSendClientPattern address)*
(C/U)	*D oneway void* **command***(in any data)*
Query	SmartQueryClientPattern (Service Requestor):
⋆	*A0 oneway void acknowledgmentConnect(in long cid,in long status)*
⋆	*A2 oneway void acknowledgmentDisconnect()*
⋆	*R3 oneway void serverInitiatedDisconnect(in long cid)*
(B/V), (D/V)	*D oneway void* **answer***(in any data,in long qid,in long status)*
	SmartQueryServerPattern (Service Provider):
⋆	*R0 oneway void connect(in SmartQueryClientPattern address,in long cid)*
⋆	*R1 oneway void discard(in SmartQueryClientPattern address)*
⋆	*R2 oneway void disconnect(in SmartQueryClientPattern address)*
⋆	*R4 oneway void requestDiscard(in SmartQueryClientPattern address,in long qid)*
(B/V), (D/V)	*D oneway void* **request***(in any data,in SmartQueryClientPattern address,in long qid)*
Push Newest	SmartPushNewestClientPattern (Service Requestor):
⋆	*A0 oneway void acknowledgmentConnect(in long cid,in long status)*
⋆	*A2 oneway void acknowledgmentDisconnect()*
⋆	*R3 oneway void serverInitiatedDisconnect(in long cid)*
(C/W)	*D oneway void* **update***(in any data,in long sid)*
	SmartPushNewestServerPattern (Service Provider):
⋆	*R0 oneway void connect(in SmartPushNewestClientPattern address,in long cid)*
⋆	*R1 oneway void discard(in SmartPushNewestClientPattern address)*
⋆	*R2 oneway void disconnect(in SmartPushNewestClientPattern address)*
⋆	*R4 oneway void subscribe(in SmartPushNewestClientPattern address,in long sid)*
⋆	*R5 oneway void unsubscribe(in SmartPushNewestClientPattern address)*
Push Timed	SmartPushTimedClientPattern (Service Requestor):
⋆	*A0 oneway void acknowledgmentConnect(in long cid,in long status)*
⋆	*A2 oneway void acknowledgmentDisconnect()*
⋆	*R3 oneway void serverInitiatedDisconnect(in long cid)*
⋆	*A4 oneway void acknowledgmentSubscribe(in long active)*
⋆	*A6 oneway void serverInformation(in double cycle,in long active)*
⋆	*R7 oneway void activationState(in long active)*
(C/W)	*D oneway void* **update***(in any data,in long sid)*
	SmartPushTimedServerPattern (Service Provider):
⋆	*R0 oneway void connect(in SmartPushTimedClientPattern address,in long cid)*
⋆	*R1 oneway void discard(in SmartPushTimedClientPattern address)*
⋆	*R2 oneway void disconnect(in SmartPushTimedClientPattern address)*
⋆	*R4 oneway void subscribe(in SmartPushTimedClientPattern address,in long rate,in long sid)*
⋆	*R5 oneway void unsubscribe(in SmartPushTimedClientPattern address)*
⋆	*R6 oneway void getServerInformation()*

Table 5.45: *The messages used internally by the communication patterns - part one.*

Pattern	Receivable Messages
Event	SmartEventClientPattern (Service Requestor):
⋆	*A0 oneway void acknowledgmentConnect(in long cid,in long status)*
⋆	*A2 oneway void acknowledgmentDisconnect()*
⋆	*R3 oneway void serverInitiatedDisconnect(in long cid)*
⋆	*A4 oneway void acknowledgmentActivate(in long status)*
(C/U), (C/W)	*D oneway void* **event***(in any data,in long aid)*
	SmartEventServerPattern (Service Provider):
⋆	*R0 oneway void connect(in SmartEventClientPattern address,in long cid)*
⋆	*R1 oneway void discard(in SmartEventClientPattern address)*
⋆	*R2 oneway void disconnect(in SmartEventClientPattern address)*
⋆	*R4 oneway void activate(in SmartEventClientPattern address,in long mode, in long aid,in any parameter)*
⋆	*R5 oneway void deactivate(in SmartEventClientPattern address,in long aid)*

Table 5.46: *The messages used internally by the communication patterns - part two.*

The *send* pattern requires the standard administrative interactions only. The *command* message of the *send* pattern carries the marshalled representation of the communication object and requires no further arguments. Since the *send* pattern does not provide any back channel, there is no need to identify the service requestor that sent the *command* message.

The (B/V) and (D/V) interaction patterns of the *query* communication pattern are composed out of the *request* and the *answer* message. The *request* contains the marshalled communication object, the address of the service requestor and the query identifier. The address is needed to be able to return the answer to the correct service requestor. The query identifier is unique at the service requestor and allows to unambiguously assign answers to pending requests and also allows to identify answers no longer needed. Thus, the query identifier is returned with the *answer* message that also contains the marshalled communication object for the answer. The status argument of the *answer* message indicates whether it contains a valid response. That is used by the server side *discard* method that can close a request without providing a valid answer in case the server is overloaded. The *request discard* is used to inform the service provider about a discarded request. It contains the address of the service requestor and the query identifier to uniquely identify the request to be discarded.

The *push* patterns require further administrative interactions to *subscribe* for updates and to *unsubscribe*. The address of the service requestor is required to identify the service requestor that is changing its subscription state. The *subscribe* message R4 of the *push timed* pattern contains another parameter which specifies to get every n-th update only. The parameter of its acknowledgment message A4 indicates whether the server is currently active. The acknowledgment A4 is not needed with the *push newest* pattern. The cycle time of the *push timed* server and its state can also be obtained by the R6/A6 interaction. The R7 message is used by the *push timed* server to inform its subscribed clients about a state change. The *update* message distributes updates to subscribed service requestors and contains the marshalled representation of the communication object and the subscription identifier that was provided by the *subscribe* message. The subscription identifier allows to identify outdated update messages.

The *event* pattern requires an event *activation* and *deactivation*. The event *activation* provides the address of the service requestor, the activation mode, the activation identifier and the marshalled representation of the communication object for the activation parameters P. The activation mode allows the service provider to already inhibit *single* activations from firing multiple times to save bandwidth. The activation identifier is unique at the service requestor and is returned with the *event* message besides the marshalled representation of the communication object E for the firing activation. It allows the service requestor to correctly assign incoming *event* messages to activations and it also allows to identify event messages not needed anymore. Of course, the *deactivation* also requires the activation identifier. The activation message R4 is acknowledged by A4 to know whether the activation was accepted by the server. Otherwise, the server would not be able to reject activations in case there is, for example, no more space to store another activation.

5.6.6.3 The Interface Objects and the Communication Pattern Interface

The generic structure of the connection between the communication patterns and the underlying communication system is shown in figure 5.80. So-called *interface objects* mediate between the communication system and the communication patterns. The task of an interface object is to map the (C⋆/U) interaction patterns onto the communication system, thereby performing all the required adjustments and conversions. The interface objects are managed by the communication patterns and are typically implemented as part of the communication patterns.

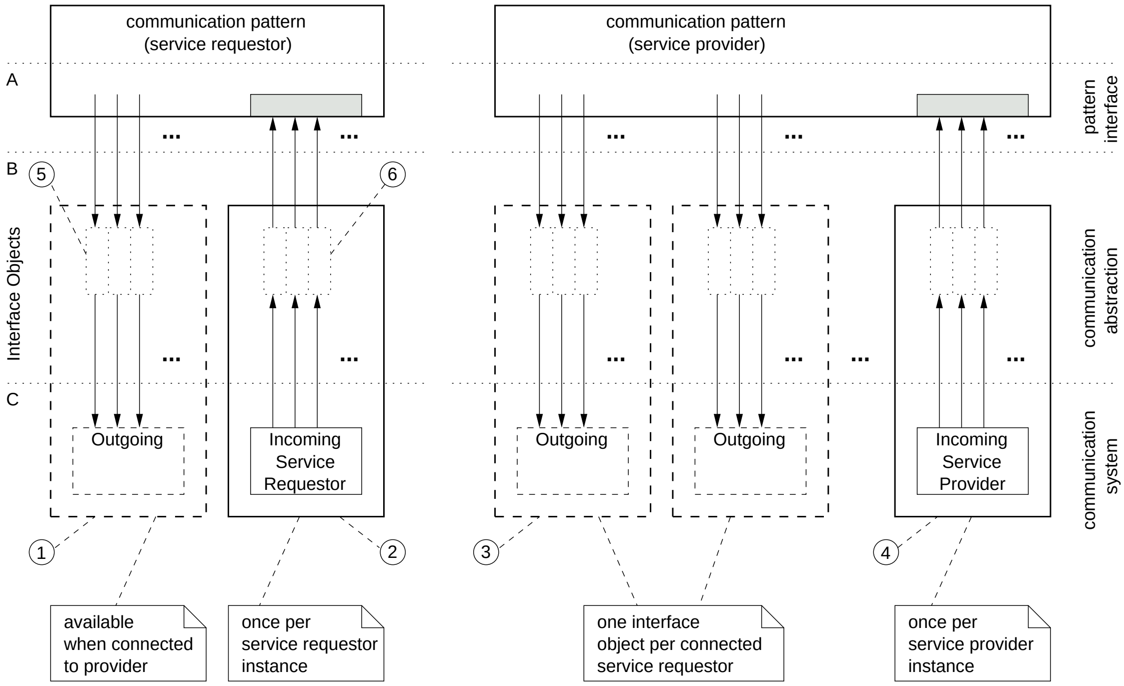

communication pattern (service requestor)
communication pattern (service provider)
A
B
C
Interface Objects
pattern interface
communication abstraction
communication system
...
5
6
Outgoing
Incoming Service Requestor
Outgoing
Outgoing
Incoming Service Provider
1
2
3
4
available when connected to provider
once per service requestor instance
one interface object per connected service requestor
once per service provider instance

Figure 5.80: *The generic structure of the connection between the communication patterns and the communication system.*

The interface object ② is generated with the creation of the service requestor. It contains an interface to the communication system with a unique address such that the service requestor can receive messages. All method arguments besides the communication objects are first demarshalled and are converted into the format that is used at the callback interface of the communication patterns ⑥. As soon as a service requestor gets connected to a service provider, it generates the interface object ① which performs all the marshalling ⑤ for the outgoing messages. The communication objects are already marshalled when they are forwarded from the communication pattern to the interface object.

The interface object ① is destroyed with a disconnect and the interface object ② gets destroyed with a destruction of the service requestor.

The interface object ④ is generated with the creation of the service provider and provides a unique address over which all messages from all service requestors are handled. The service provider generates a separate interface object ③ for each connected service requestor. The interface objects ③ are destructed as soon as the corresponding service requestor gets disconnected. The interface object ④ gets destructed with the shutdown of the service provider.

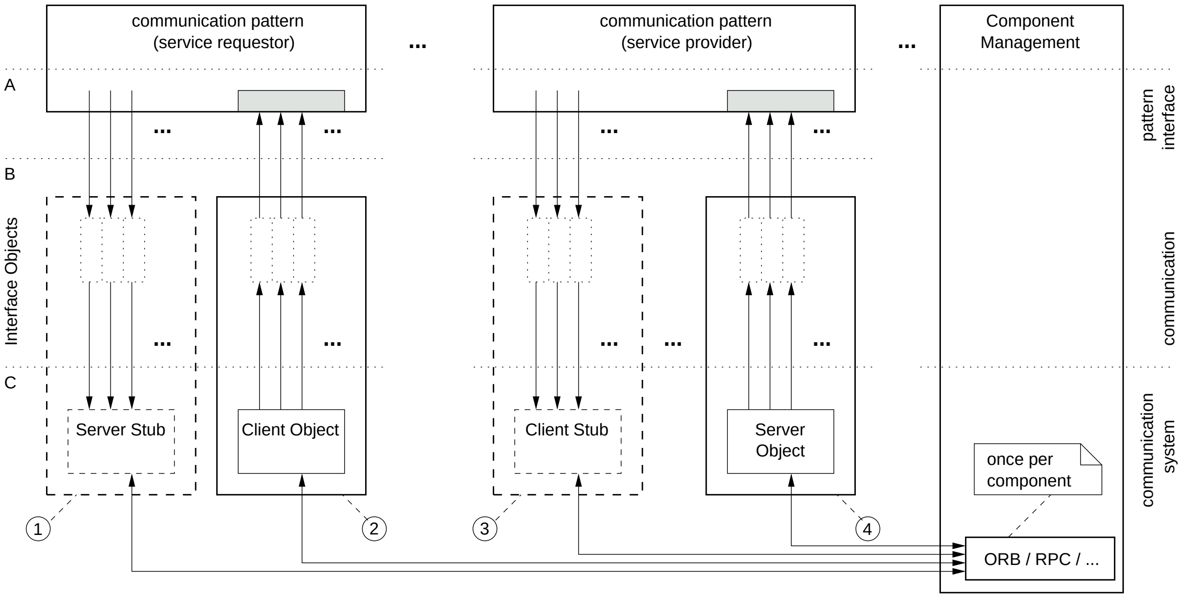


Figure 5.81: *The interface objects of the communication patterns with an object based middleware.*

Figure 5.81 illustrates the interface objects with an object based middleware like *CORBA* or *remote procedure calls*. Each message is represented by a member function of a remotely callable object. Sending a message is calling a member function of a remote object via the appropriate stub and receiving a message corresponds to executing the corresponding member function at the servant object. The implementations of the methods of the servant objects provide the required adjustments and invoke the upcall interface of the communication pattern. The interface objects for outgoing messages provide the same methods as the stubs, perform the adjustments of the parameters and invoke the corresponding method of the stub.

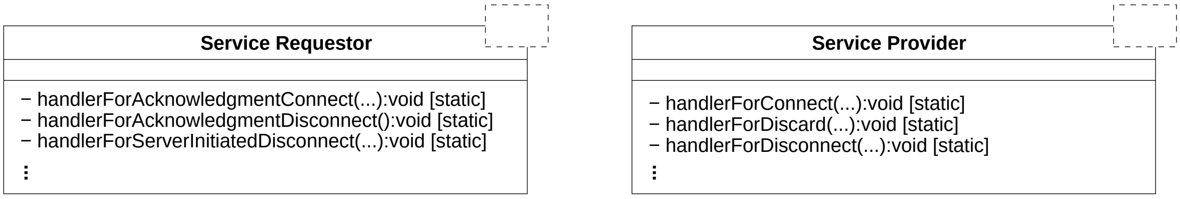

Service Requestor
− handlerForAcknowledgmentConnect(...):void [static]
− handlerForAcknowledgmentDisconnect():void [static]
− handlerForServerInitiatedDisconnect(...):void [static]
⋮
Service Provider
− handlerForConnect(...):void [static]
− handlerForDiscard(...):void [static]
− handlerForDisconnect(...):void [static]
⋮

Figure 5.82: *Implementation details of the upcall interface of the communication patterns.*

Figure 5.82 shows some details of the implementation of the upcall interface of the communication patterns. The communication patterns provide handlers for all incoming messages. The arguments of the handler functions correspond to the arguments of the messages. However, the argument list is extended by a *this*-pointer to the communication pattern instance as first argument. The *void*-casted *this*-pointer to the communication pattern instance as well as the addresses of its callback methods are provided with the constructor of the interface object. The *this*-pointer is needed inside the handler to grant access to the communication pattern instance since the upcall handlers are implemented as *static* member functions. The reason for this are more generic interface objects that do not depend on the template bindings of the communication patterns. That definitely simplifies the handling of pointers to callback methods. With the *CORBA* based implementation, the *server* and the *client* objects with their corresponding stubs are automatically generated from the *IDL* description of the receivable messages as illustrated in figure 5.83.

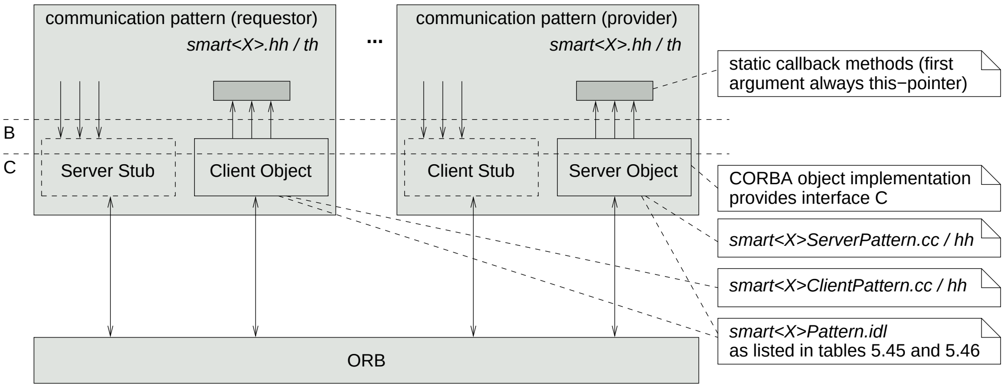

communication pattern (requestor)
smart<X>.hh / th
...
communication pattern (provider)
smart<X>.hh / th
static callback methods (first argument always this−pointer)
B
C
Server Stub
Client Object
Client Stub
Server Object
CORBA object implementation provides interface C
smart<X>ServerPattern.cc / hh
smart<X>ClientPattern.cc / hh
smart<X>Pattern.idl as listed in tables 5.45 and 5.46
ORB

Figure 5.83: *The interface objects with* CORBA.

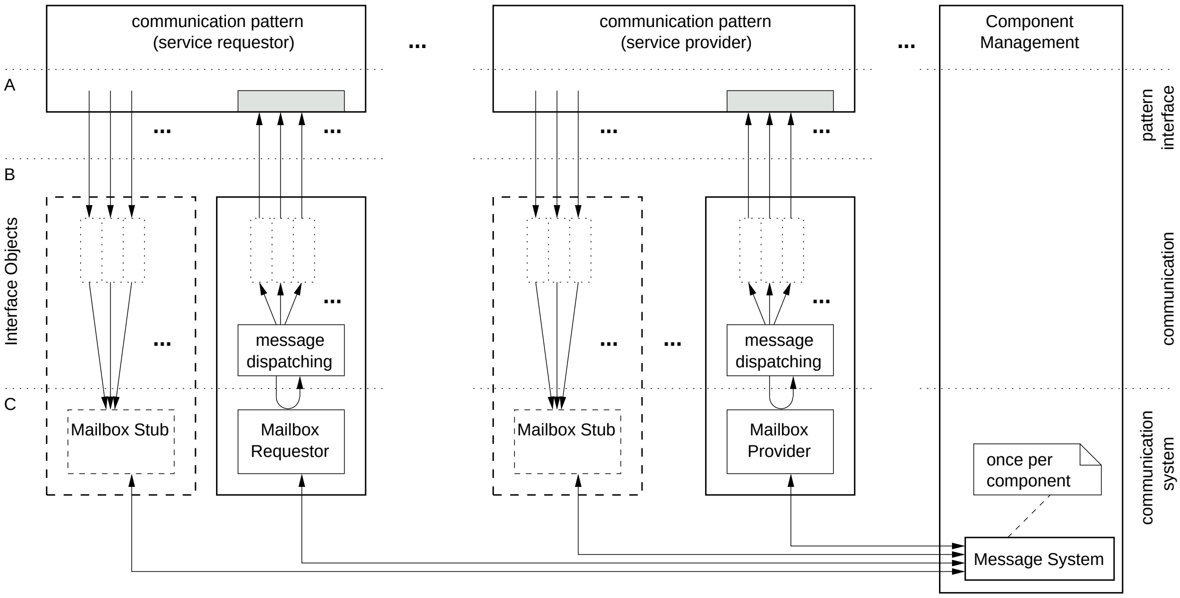


Figure 5.84: *The interface objects of the communication patterns with a message based middleware.*

Figure 5.84 illustrates the interface objects with a message based communication system that can be either a mailbox based communication system or even a socket based communication mechanism. In contrast to the object based approaches, one now needs a message dispatching mechanism. A message based approach is normally based on a reactor pattern [138] to accept incoming messages and to dispatch and forward them to the appropriate handlers of the communication patterns.

The servant object respectively the mailbox or the socket that is contained in the interface objects ② and ④ is shared by all interaction patterns of a communication pattern. The reasons why this does not impose any restrictions and what kind of threading models are needed within the interface objects is explained after the locking mechanisms and their interactions are introduced in sections 5.6.6.6, 5.6.6.7 and 5.6.6.8.

5.6.6.4 The Addressing Scheme

The internally used addressing scheme has to implement the naming of services described in section 5.4.5. The overall scheme is presented in figure 5.85. The naming service operates at the level of services and translates the general naming scheme into addresses for direct communication. Names of services have to be unique only within the scope of a component since the name of a service is always additionally tagged by the name of the component. The user visible naming scheme addresses services by a tuple {*component name, service name*} consisting of the user provided name of the component and the user provided name of the service. Unique names of services are necessary to distinguish different services composed of the same communication pattern instantiated by the same communication objects. Inside a communication pattern, this tuple is extended by {*pattern type*} followed by any number of *names of communication objects*. The pattern type identifier is either *send*, *query*, *push newest*, *push timed* or *event* and denotes the communication pattern type. As explained below in section 5.6.7.7, the *wiring* pattern internally uses the *query* pattern and does therefore not constitute its own pattern type. The list of communication object names comprises the user defined names of the communication objects that bind the communication pattern template. The name of a communication object is provided to the communication pattern by the *name* member function of the framework interface of a communication object. A service provider uses the extended name to register its address at the name service. A service requestor can get the address of a service provider by composing the same extended name and asking the name service for the corresponding address. The name service only returns the address of a compatible service since otherwise the extended name used to look up the address was already composed wrongly. If the name service has no record of the provided extended name then there is no such service provider available.

The used naming scheme ensures that the name service only returns addresses of compatible service providers. Compatibility requires a service provider to match the type of the communication pattern and the names of the involved communication objects. Since those are coded in the extended name, the name service can only find compatible services. A service requestor can therefore never connect to an incompatible service provider and a component always only receives messages of expected types. On top of the object based communication systems, this makes sure that the stub object always matches its implementation inside its communication pattern counterpart.

Service requestors do not need to be made public and are therefore not registered at the name service. Instead, each *connect* of a service requestor provides its address to the service provider in order to receive messages that are sent back from the service provider. Thus, a service requestor has its private address and additionally holds the address of a service provider if it is connected to one. A service provider possesses a public address and holds a list of the addresses of its connected service requestors.

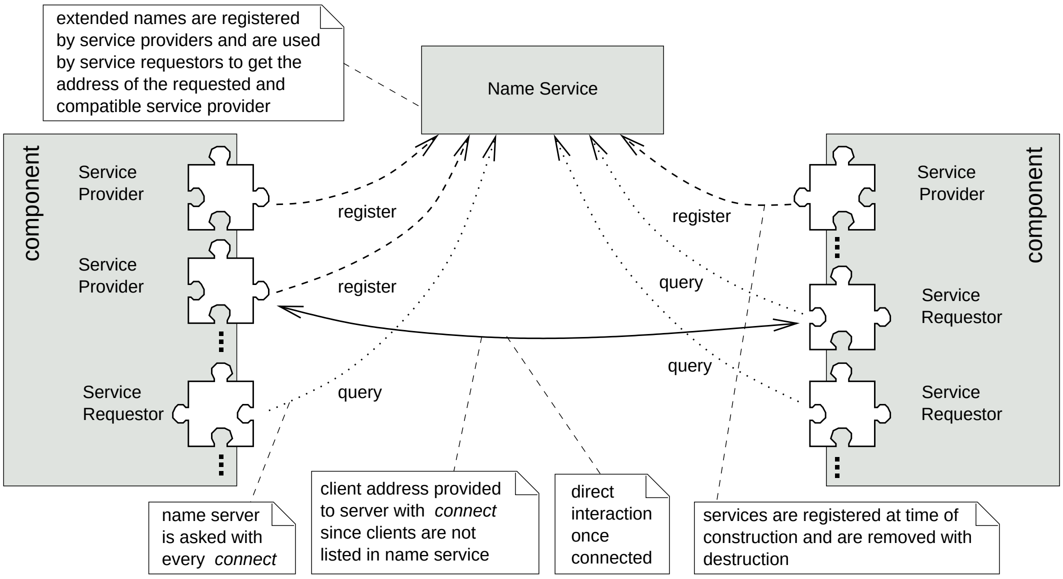

extended names are registered by service providers and are used by service requestors to get the address of the requested and compatible service provider
Name Service
component
Service Provider
Service Provider
Service Requestor
register
register
query
register
query
query
Service Provider
Service Requestor
Service Requestor
component
name server is asked with every *connect*
client address provided to server with *connect* since clients are not listed in name service
direct interaction once connected
services are registered at time of construction and are removed with destruction

Figure 5.85: *The general addressing scheme.*

Object Based Middleware The general addressing scheme on top of an object based middleware is illustrated by means of *CORBA*. With an object based middleware, there always exists a one-to-one relation between a servant object and its *object reference*. The *object reference* provides the unique address of the servants inside the interface objects ② and ④ shown in figure 5.81 and it already represents the fully decoded extended name since each message type is mapped to a separate member function. Possessing the *object reference* allows to access the member functions of the corresponding servant and calling one of the servant's member functions corresponds to sending that message to the proper component and service. The implementation of the addressing scheme on top of object based middleware is summarized in figure 5.86.

The service provider registers the reference of the servant object of its interface object ④ at the name service using the extended naming scheme. The object reference needed by the interface object ③ is obtained from the service requestor with a *connect*. A *connect* transmits the object reference of the client object of ② to the service provider. The object reference needed by the interface object ① is obtained from the name service when performing a *connect* to a service provider.

The chosen implementation of the general addressing scheme allows neither to connect to incompatible services nor to send inappropriate messages from a pattern to its opponent. The first property is a result of having access to middleware objects only that either have been provided by the name service and thus represent a compatible service provider or that have been provided with the *connect* and thus represent a compatible service requestor. The second property holds true due to the fact that all member functions correspond to valid messages and that other member functions are not available. Furthermore, messages can never be delivered to an old instance of a servant object since each newly started service requestor and service provider possesses a new servant object with a unique identity. Normally, that feature holds true with every object based middleware. Thus, for example, destroying a service provider and restarting it very fast is always detected since the object reference is not valid anymore. Therefore, one never wrongly continues to interact after all the states in either the service provider or the service requestor were lost. The new instance, of course, has a different address but is

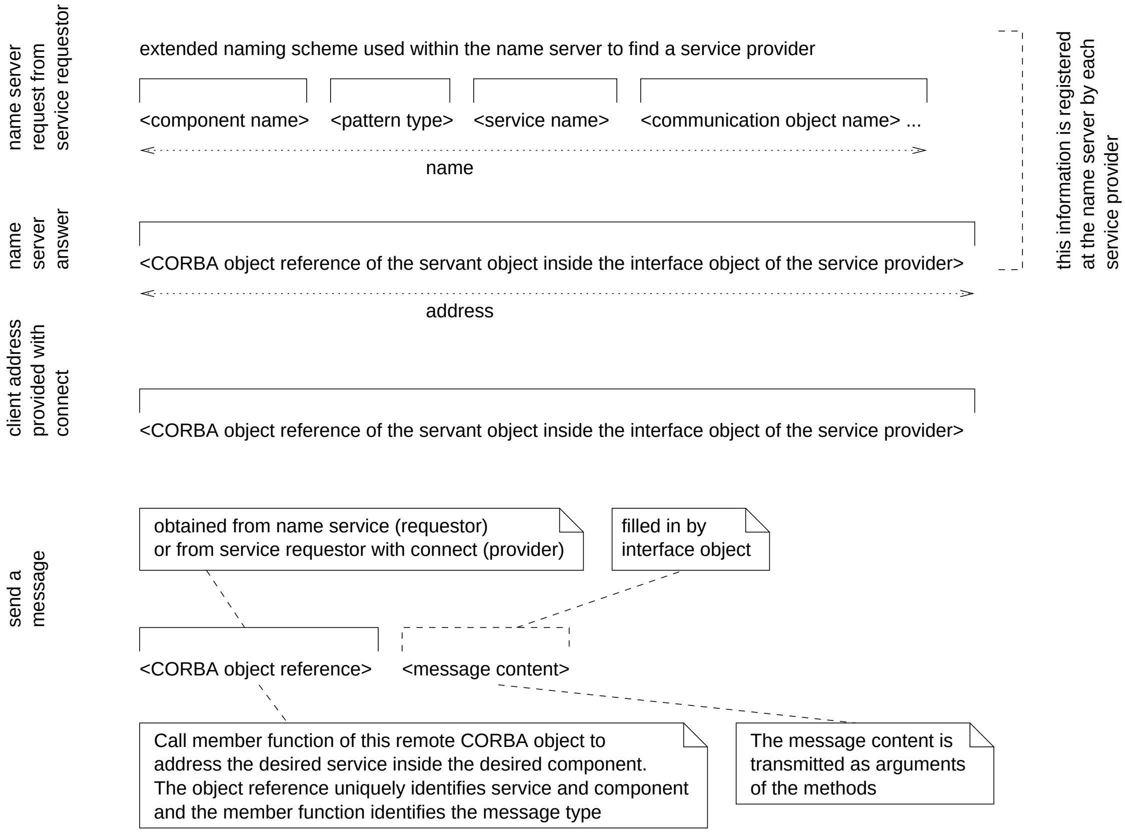

name server request from service requestor
extended naming scheme used within the name server to find a service provider
<component name>
<pattern type>
<service name>
<communication object name> ...
name
name server answer
<CORBA object reference of the servant object inside the interface object of the service provider>
address
this information is registered at the name server by each service provider
client address provided with connect
<CORBA object reference of the servant object inside the interface object of the service provider>
send a message
obtained from name service (requestor) or from service requestor with connect (provider)
filled in by interface object
<CORBA object reference>
<message content>
Call member function of this remote CORBA object to address the desired service inside the desired component. The object reference uniquely identifies service and component and the member function identifies the message type
The message content is transmitted as arguments of the methods

Figure 5.86: *The addressing scheme on top of* CORBA.

registered with the same extended name at the name service.

Message Based Middleware The general addressing scheme on top of message based middleware is illustrated by means of *TCP sockets*. Again, the entries at the name service are organized such that only addresses of compatible services are returned. In contrast to the object based middleware systems, the process of dispatching messages is now visible to the framework builder. Thus, the address comprises two extensions as shown in figure 5.87.

The first extension is the *service identifier*. After a socket address got released, it can be reused by another interface object on the same host. Thus, the socket address is not sufficient to unambiguously identify an interface object and one needs an identifier that is unique host-wide. Some operating systems already provide a system call to generate unique identifiers. In all other cases, the concatenation of the memory address of the interface object and its creation time is a suitable identifier.

The second extension is the *message type* that identifies the handler to be called inside the interface object. The combined evaluation of the *service identifier* and the *message type* uniquely identifies the correct pattern instance and message handler.

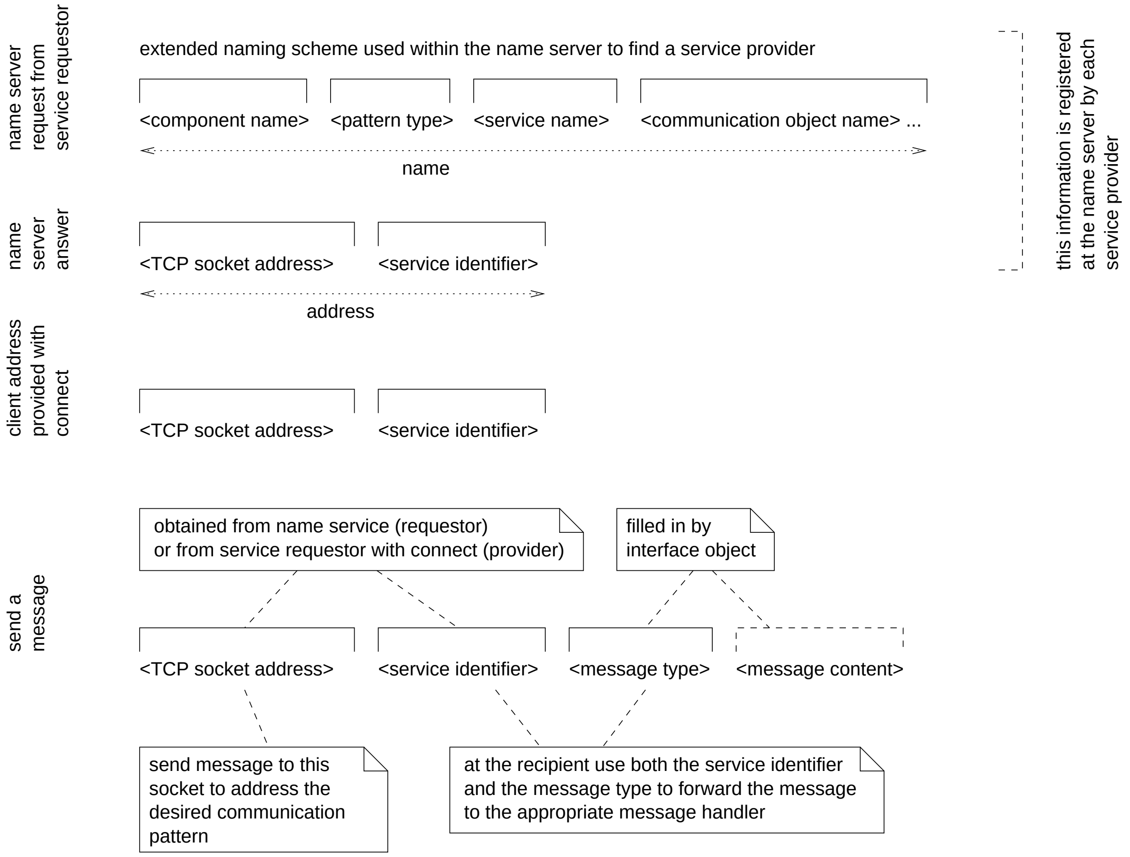

name server request from service requestor
extended naming scheme used within the name server to find a service provider
<component name>
<pattern type>
<service name>
<communication object name> ...
name
name server answer
<TCP socket address>
<service identifier>
address
client address provided with connect
<TCP socket address>
<service identifier>
this information is registered at the name server by each service provider
send a message
obtained from name service (requestor) or from service requestor with connect (provider)
filled in by interface object
<TCP socket address>
<service identifier>
<message type>
<message content>
send message to this socket to address the desired communication pattern
at the recipient use both the service identifier and the message type to forward the message to the appropriate message handler

Figure 5.87: *The addressing scheme on top of* TCP sockets.

5.6.6.5 The Pattern Internal Handling of Blocking Calls

Interaction patterns require means to efficiently handle blocking method based interfaces that wait for an outstanding answer or that await an asynchronous notification. Blocking calls are not allowed to waste processing time during blocking and blocking calls have to be abortable. Both, the assignment of responses to open requests and the management of whether blocking is allowed is based on a state automaton. Thus, one requires means to coordinate concurrent access to the state automaton and to properly notify suspended calls about performed state transitions.

All those requirements can be met by using a *monitor* [138]. The monitor coordinates and protects access to the state automaton. A monitor is also known as thread-safe passive object since a monitor does not have its own thread of execution. This makes a monitor very resource friendly. With a monitor, no processing time is needed when awaiting a specific state of the state automaton. A monitor allows to perform state changes while other threads are blocked on the state automaton. Concurrent requests from multiple threads are serialized and require no further arrangements outside the monitor.

A monitor is needed for those parts of the interaction patterns that provide a user invoked method that gets blocked until a request or a response is available. Thus, a monitor is required for the client part characteristics (B) and (D) and the server part characteristic (W). With the (B/U), (B/V), (D/U) and (D/V) interaction patterns, the monitor is placed at the client part. It awaits the response message and ensures that the blocked client side method can be aborted. In case of the (D/U) and (D/V) interaction patterns, the blocking method is the *response* method. With the (C/W) interaction pattern, the monitor is placed at the server part to manage the blocking *invocation* method.

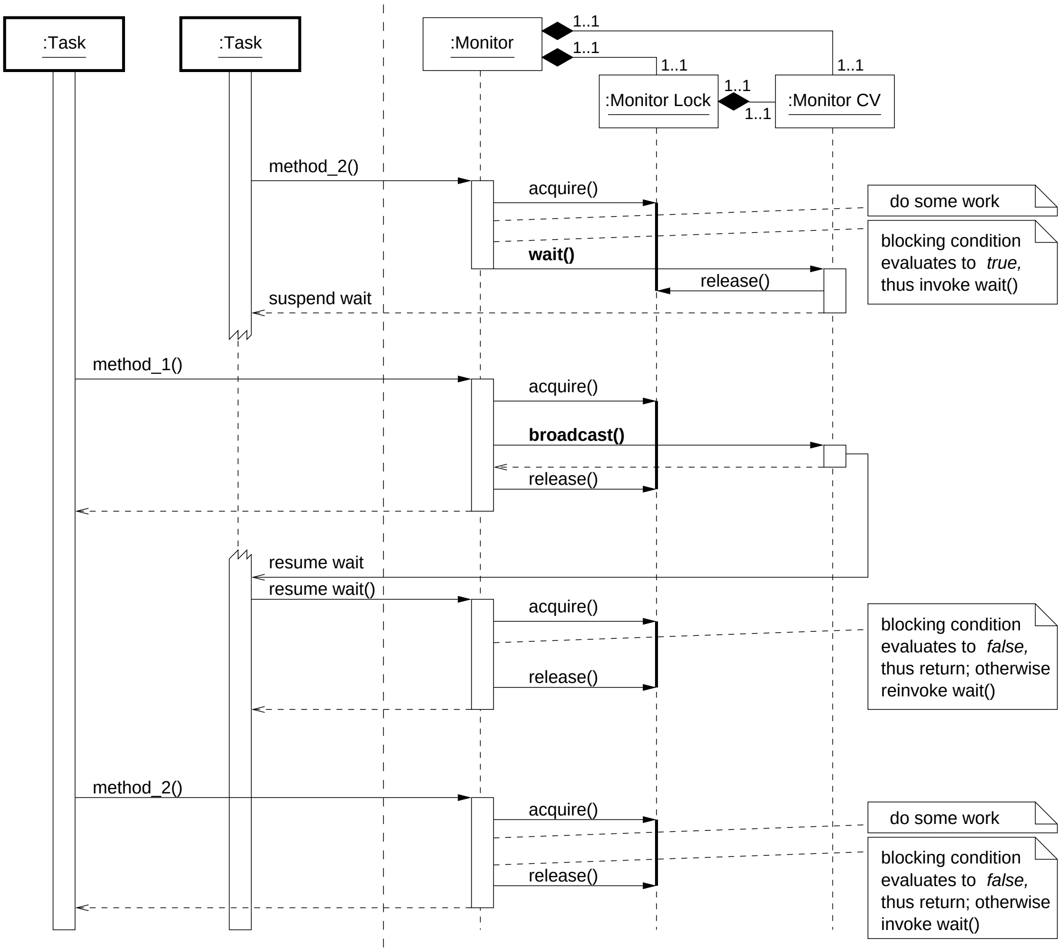


Figure 5.88: *The interaction of the user member functions with the monitor class.*

The general scheme of a monitor is shown in figure 5.88. According to [135], a monitor synchronizes method execution to ensure that only one method runs within an object at a time and thus makes sure that simultaneously accessing the object by two or more threads causes no conflicts. This is achieved by a *monitor lock*. Each synchronized method first acquires the lock before performing any activity and releases the lock just before leaving the method. A thread is blocked when it calls a monitor method while another thread has already entered the monitor. The blocked thread is woken up when the active thread exits the monitor. The monitor lock is based on a *recursive thread mutex* which allows recursive calls to synchronized member functions.

A monitor schedules its synchronized methods cooperatively. Synchronized methods use monitor conditions to determine the circumstances under which they should suspend or resume their processing. If a synchronized method must block, it can wait on one of its monitor conditions which of course requires to temporarily leave the locked section [69]. This automatically releases the monitor lock and allows other threads to proceed while waiting.

A synchronized method of a monitor can notify one or all methods waiting for the monitor condition. A notified method automatically reacquires the lock and resumes execution. Normally, one first checks the new state of the monitor condition and decides whether to wait again or to proceed. Leaving the method releases the reacquired lock. In case all waiting methods got notified, one after the other is executed and is synchronized by the lock. The notification mechanism and the evaluation of the monitor condition is based on a *condition variable* whose mutex is the monitor lock. Waits therefore block on the condition variable and changes to shared data that are protected by the monitor are announced by a signal on the condition variable. A signal is effective to currently waiting calls only and is not memorized. Everything to be memorized is stored in the state automaton. The condition variable ensures that no state change is missed between testing the state automaton and getting suspended by the wait member function.

A standard monitor signals every state change and always wakes up all suspended threads. All threads can test their individual blocking condition. They just reinvoke the *wait* if the state is not yet reached that is required to resume processing. Thus, with every state change, a lot of time consuming task switches are wasted for those threads that can still not continue with their work and that again get suspended.

In contrast thereto, the presented monitors signal only those state changes that actually allow a suspended member function to resume its processing. This avoids unnecessary task switches and improves the overall performance. The test on whether to wake up a *wait* is shifted from the individual blocking condition of the *wait* towards those member functions that perform a state transition. However, that is possible only if no blocking member function requires an individual blocking condition. It makes sense only if a once resumed *wait* never needs to be reinvoked. Both preconditions hold true for all the monitors that are required inside the communication patterns and these properties are illustrated in detail with the more complex monitors described in section 5.6.7.

In case of the monitors of the communication patterns, member functions of a monitor get blocked on the state automaton only to await the next message of the monitored interaction. The crucial point is that all concurrent and blocked calls of the monitor are related to the very same interaction. They all see the same singular state automaton and all concurrent and blocked calls await the very same message. Thus, once that message arrived, further blocking is obsolete for all the blocked calls. Therefore, once a blocked call is resumed due to having reached the awaited state, it never reinvokes the *wait* again. Since all blocked calls work on the same state automaton and await the same message, they are all released in the same way and independently of an individual blocking condition.

Blocked calls are also awakened independently of an individual blocking condition in case that a transition is performed into another state than the awaited one. Since all suspended calls are related to the very same interaction, they all have to get informed about the newly reached state which requires to resume all the *waits*. However, the crucial point now is that a once resumed *wait* never needs to be reinvoked even though the blocked member function got resumed in the not awaited state. That holds true since the state automatons of the monitors of the communication patterns are naturally structured such that getting resumed in a state that is different to the awaited one is always either tantamount to the case that the awaited message cannot be received anymore or that it is obsolete or is tantamount to the case that no blocking is allowed anymore. In all cases, one does not need to reinvoke the *wait* from inside the method call and thus again, no individual blocking conditions are required.

The outcome of this is the following rule of when to invoke a broadcast on the condition variable of the monitor. A *wait* blocks in those states only that can still expect the awaited message. All transitions from blocking to non-blocking states have to broadcast a signal. That is since a blocking method would otherwise not be able to follow the state transition into the non-blocking state. For state transitions leading from a blocking state into the same or another blocking state, only those transitions emit a broadcast that correspond to the arrival of a message. Even if one awakes in a blocking state, the arrival of a message denotes that all pending methods have to return. For example, one can awake in the same blocking state when waiting for the next update. Of course, one has to return with the update instead of reinvoking the *wait*.

An important issue with condition variables is their exact semantics. Normally, condition variables are assumed to implement a Hoare-style semantics. That means that the signaller gives up control immediately to the thread being woken up. However, nearly all practical implementations of condition variables provide a Mesa-style semantics only since that is much easier to implement. The important difference is that with a Mesa-style semantics, the woken up thread is simply put on the ready list. The consequence of using a Mesa-style semantics is that some other thread can acquire the lock and can change data structures after the *wait* got signalled and before it resumes processing. The problem then is that the awaited condition might not be *true* anymore and that one normally would decide to reinvoke the *wait*. Although the awaited state was *true* meanwhile, it can thus be missed. Of course, the presented monitors can also miss states but since they never reinvoke a *wait* they never miss the information that the awaited state was reached. Thus, the signalling policy used for the monitors of the interaction patterns circumvents the drawbacks of the Mesa-style implementation since one never reinvokes a *wait*.

The Monitor Base Class Figure 5.89 shows the class diagram of the monitor base class. It provides the basic infrastructure only and derived monitor classes solely add individual data structures that need to be protected by the monitor. In contrast to standard monitors, the *acquire* and *release* methods provide access to the monitor lock and the *wait* and *broadcast* methods provide access to the monitor condition variable. That allows to apply the monitor without shifting all the activities to be coordinated into a member function of the monitor. In particular with the monitors of the interaction patterns, this makes it much easier to access the infrastructure of the communication pattern from inside an activity that needs to be protected by a monitor. Otherwise, one would have to provide appropriate references to the communication pattern infrastructure with each method of the monitor. Of course, prior to calling the *wait* respectively the *broadcast* member function, one has to *acquire* the monitor lock and after returning from these functions, one has to *release* the lock.

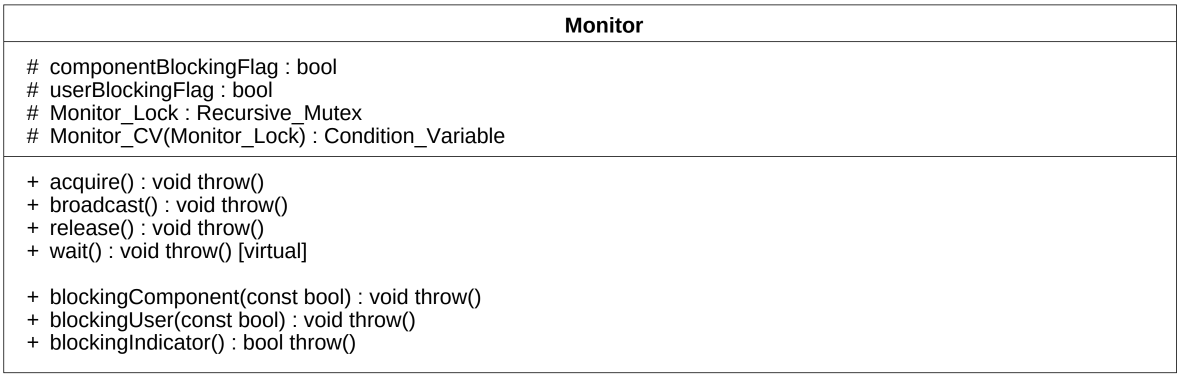

Monitor
# componentBlockingFlag : bool
# userBlockingFlag : bool
# Monitor_Lock : Recursive_Mutex
# Monitor_CV(Monitor_Lock) : Condition_Variable
+ acquire() : void throw()
+ broadcast() : void throw()
+ release() : void throw()
+ wait() : void throw() [virtual]
+ blockingComponent(const bool) : void throw()
+ blockingUser(const bool) : void throw()
+ blockingIndicator() : bool throw()

Figure 5.89: *The class diagram of the monitor base class.*

Common to all monitors are the two separate blocking flags that are combined by a logical *and* to form the blocking indicator. Blocking is allowed only if the blocking indicator evaluates to *true*. The *user blocking flag* is used by the *blocking* member function that is part of the user interface of each service requestor part of a communication pattern. The *component blocking flag* is used to implement the blocking mode of the component management that is effective component wide. Both blocking flags are initialized to *true*. As illustrated in figure 5.90, the methods to modify the blocking flags and those to get the state of the blocking indicator are already protected by the monitor lock. Independently of the individual state automaton, all currently blocking *waits* are aborted and all further calls to the *wait* member function return immediately and are not blocked as soon as the blocking indicator evaluates to *false*. A broadcast is emitted only if the state of the blocking indicator switches from *true* to *false* since that is the only case where already blocked methods have to be aborted. A broadcast is not required with all other state transitions since then blocking either is allowed or has already not been allowed. In the latter case, there are no pending *waits* since they all were aborted with the state change to the *false* state and since no new *waits* are invoked as long as the blocking indicator evaluates to *false*.

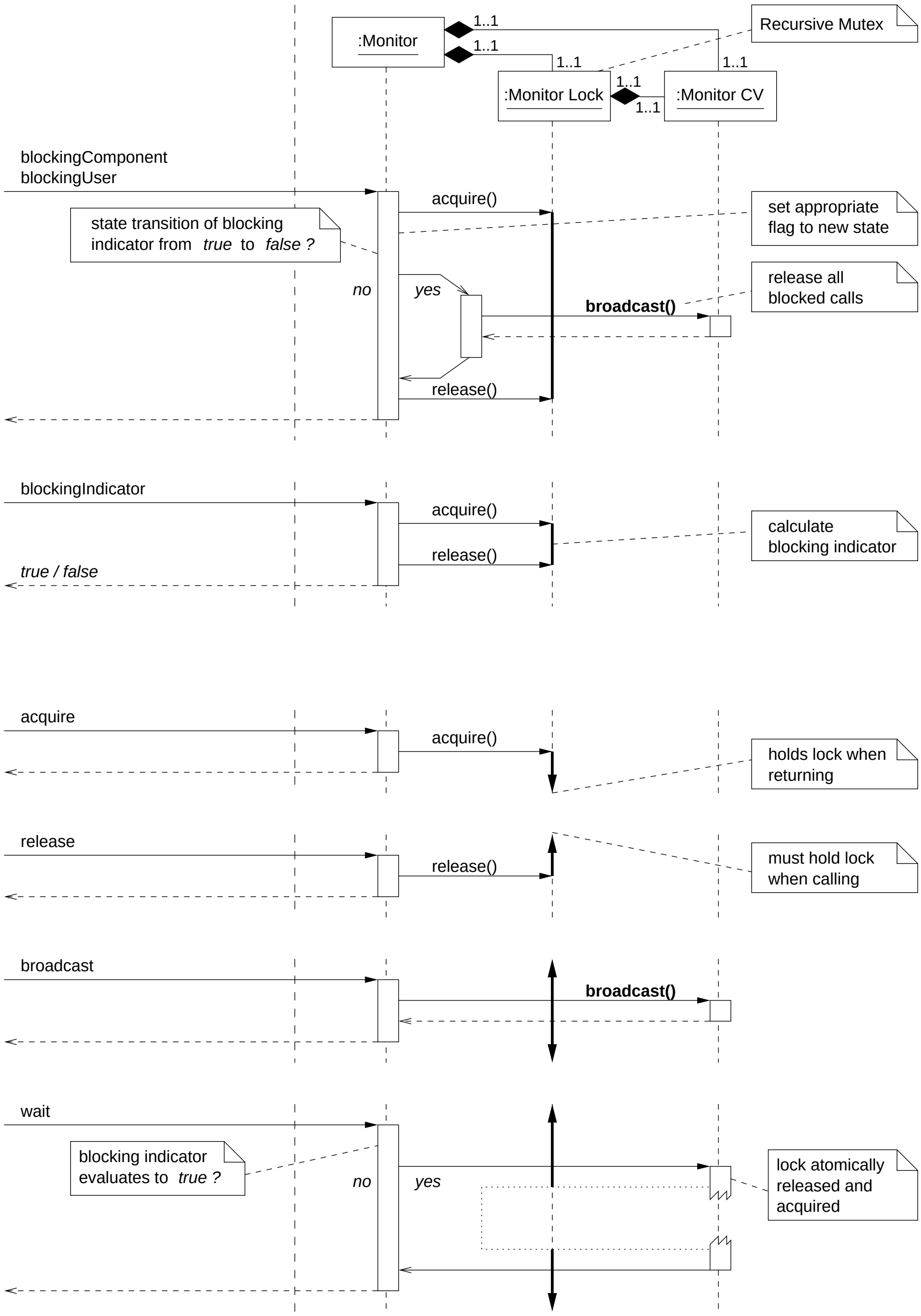

:Monitor
1..1
1..1
1..1
1..1
Recursive Mutex
:Monitor Lock
1..1
1..1
:Monitor CV
blockingComponent
blockingUser
acquire()
state transition of blocking indicator from true to false ?
set appropriate flag to new state
no
yes
broadcast()
release all blocked calls
release()
blockingIndicator
acquire()
calculate blocking indicator
release()
true / false
acquire
acquire()
holds lock when returning
release
release()
must hold lock when calling
broadcast
broadcast()
wait
blocking indicator evaluates to true ?
no
yes
lock atomically released and acquired

Figure 5.90: *The sequence diagrams of the methods of the monitor base class.*

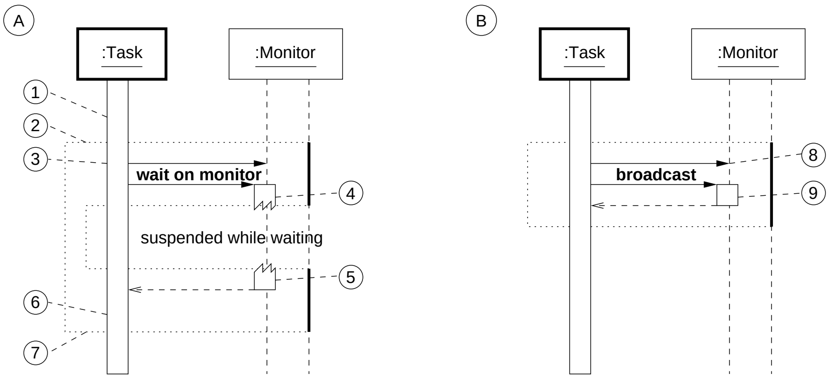


Figure 5.91: The simplified representation of applying the monitor.

Figure 5.91 shows the simplified representation of a monitor that is henceforth used in the sequence diagrams. A task performs some work ① and acquires the monitor lock by calling the *acquire* method ② as soon as it enters the scope of the monitor. The scope of the monitor is entered either when accessing data structures that are protected by the monitor or when entering a critical section that can be entered only under the supervision of the monitor. As soon as the monitor lock is acquired, no concurrent state change can occur and no other method belonging to the scope of the monitor is executed and one can safely operate on the monitor protected data structures ③. Calling the *wait* atomically releases the lock ④ and suspends the task. As soon as the monitor gets signalled, the task is resumed and reacquires the lock ⑤. Again, all other activities that enter the scope of the monitor are blocked meanwhile ⑥ and are resumed earliest after the lock is released ⑦. Of course, in ④, one checks whether to invoke the *wait* and in ⑤, one checks the current state in which one ended up. With the monitors used by the interaction patterns, one never reinvokes the *wait* in ⑤. The figure on the right shows the simplified representation of the standard sequence of entering the scope of the monitor. One first acquires the lock to be allowed to continue and to be allowed to access and to modify ⑧ the protected data structures. If a state change requires to release blocked calls, one can emit a broadcast ⑨ before one releases the lock.

The Simplified Monitor of the Administrative (B/U) Interaction Administrative interactions that are based on the (B/U) interaction pattern require a monitor to implement the client part characteristic (B). However, they can already do without an elaborated state automaton. A mutex inside the communication pattern already ensures that always only one administrative interaction is active. Furthermore, an administrative interaction can always be completed and does not need to be abortable. Thus, no complex state automaton is required with that monitor.

The simplified monitor of the administrative (B/U) interactions is shown in figure 5.92. Prior to sending the request, one invokes the *prepare* method to set an identifier that allows to uniquely identify the awaited response. It is important to note that sending the request is not protected by the monitor lock. As explained below, not holding the monitor lock while sending the request is the key why even on top of a synchronous communication mechanism, the administrative interactions do not require further decoupling mechanisms. After having sent the request, one invokes the *wait* method that either returns after a timeout occurred or after the expected response arrived. The timeout is solely used with the *connect* of the service requestor. The administrative interactions do not use the blocking flags of the monitor. The upcall, that provides the response, acquires the monitor lock and checks the

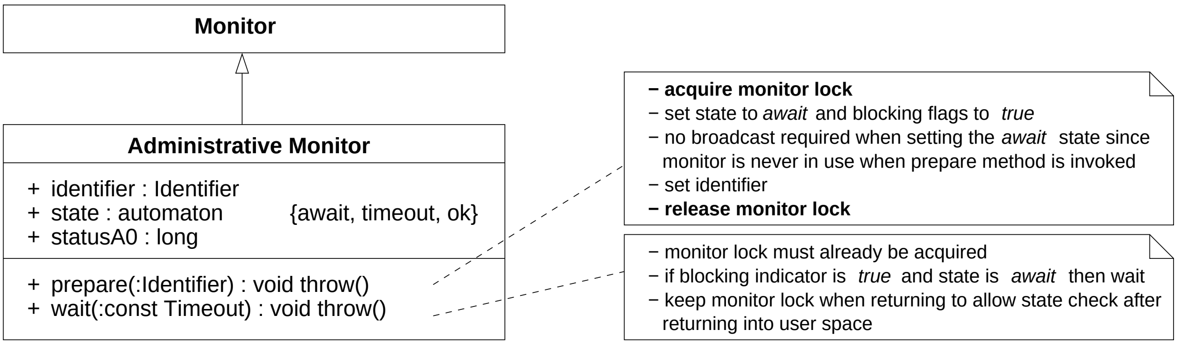

Monitor
Administrative Monitor
+ identifier : Identifier
+ state : automaton {await, timeout, ok}
+ statusA0 : long
+ prepare(:Identifier) : void throw()
+ wait(:const Timeout) : void throw()
− acquire monitor lock
− set state to await and blocking flags to true
− no broadcast required when setting the await state since monitor is never in use when prepare method is invoked
− set identifier
− release monitor lock
− monitor lock must already be acquired
− if blocking indicator is true and state is await then wait
− keep monitor lock when returning to allow state check after returning into user space

***Figure 5.92:** The class diagram of the simplified monitor used inside the administrative (B/U) interaction pattern. It is derived from the monitor base class.*

identifier. If the identifier matches the expected one, the upcall sets the state automaton to *ok*, stores the received arguments such that they are protected by the monitor and emits a broadcast. Finally, the monitor lock is released. An administrative interaction never receives more than one response and thus no response can get missed due to overwriting. Due to the unique identifier, no late answer can mess up an interaction and since the identifier is set prior to sending the request, one can never miss a response that arrives before the *wait* is invoked even when the monitor lock is released before the *wait* is invoked.

The administrative monitor also holds the variables that are needed to handover returned values from the response message to the communication pattern. The *statusA0* attribute is used by the A0 message to return the status whether the *connect* was accepted or not.

5.6.6.6 The Locking Strategies of the Communication Patterns

A crucial role inside the communication patterns is played by the locking mechanisms. These have to ensure that no deadlocks can occur and that all concurrently active interaction patterns never interfere. The internals of an interaction pattern are coordinated by a monitor. However, the *connection oriented split protocol* additionally requires mechanisms to coordinate the administrative interactions and to coordinate them with the interaction patterns that implement the actual service. Only the proper interaction of the administrative interaction patterns with the service related interaction patterns ensures that the latter show the specified characteristics even on top of (C*/U) interaction patterns.

The locking mechanisms are designed such that the administrative interactions from a service requestor to a service provider do not depend on any decoupling. Thus, they even work on top of a *processed* policy and a *delivered* policy needs not to be emulated for them in case that only a *processed* policy is available which makes their implementation very efficient. The locking mechanisms are introduced on top of a *delivered* policy and are then reconsidered on top of a *processed* policy.

The Administrative (B/U) Interaction The overall locking strategy is explained by means of the administrative (B/U) interaction. As already mentioned, all administrative interactions are serialized such that they are never active concurrently. The serialization is mandatory since it makes no sense, for example, to invoke a *subscribe* while an *unsubscribe* or a *disconnect* is still active. The difference between administrative and service related interactions is that always only one administrative interaction is active. Thus, both types of interactions differ with respect to releasing locks.

Figure 5.93 illustrates an administrative (B/U) interaction from the service requestor to the service

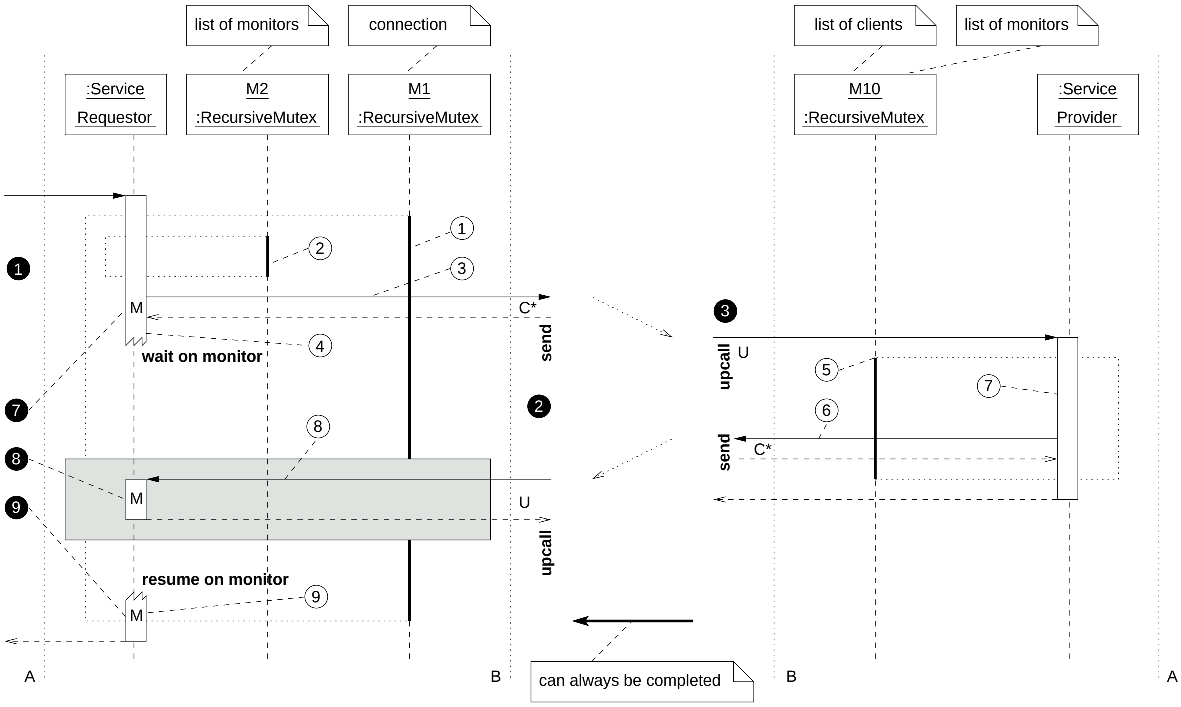


Figure 5.93: *Administrative (B/U) interaction from the service requestor to the service provider (*delivered *policy).*

provider. An M in the activation box of the service requestor denotes the access to the monitor of the administrative interaction. The top most M denotes the *prepare* method that is invoked prior to the *send* ❼. The M in the upcall ❽ is the simplified representation of the sequence denoted by B in figure 5.91. Finally, ❾ is the simplified representation of releasing the monitor lock after it is not needed anymore. The monitor lock was acquired automatically with resuming the activity after the *wait*.

At the service requestor part of a communication pattern, the mutex M1 protects the connection related status flags like the *connected* and the *subscribed* flag and it serializes the administrative interactions so that these cannot get interleaved and mess up the connection status. Thus, every administrative interaction must always hold the mutex M1 while being executed. Of course, the mutex M1 can also be acquired to inhibit any connection related changes since no administrative interaction gets executed then. Holding the mutex M1 protects the connection to the service provider from being modified while an interaction pattern of the service requestor accesses the service provider. The mutex M10 performs the analogous task at the service provider part of a communication pattern. It must be hold either if a connected service requestor is accessed or if the list of connected service requestors is modified.

Since always only one administrative interaction can be active, the administrative (B/U) interactions are based on a static monitor instance that could even be shared by all administrative interactions. In contrast thereto, a service related interaction can be invoked by any number of concurrent threads. For example, each not yet completed *query* can be seen as a separate instance of the appropriate interaction pattern. Of course, all *queries* are handled by the same interaction pattern instance but they all possess their own monitor instance. Thus, the monitor instances of the service related interactions are generated dynamically. The administrative interactions always have to be able to notify the currently active service related interactions about connection related state changes. Therefore, each communication pattern that has to handle dynamically generated monitor instances, possesses a *list of monitors*. At the service requestor the *list of monitors* is protected by the mutex M2. At the service provider, the mutex M10 also undertakes the task of protecting the *list of monitors*. For reasons that get clear with the subsequent explanations on the locking strategy, the mutex M1 cannot be used to protect the *list of monitors* at the service requestor.

The dynamically generated monitor instances have to be managed by *smart pointers* [11, 36]. The reason is that several threads can be blocked on one monitor instance and one never knows when all threads are released such that the monitor instance is not needed anymore. That problem is further intensified by the Mesa-style semantics of the condition variables. With *smart pointers*, the dynamically generated monitor instance is destroyed automatically as soon as the last shared pointer pointing to the monitor instance is deleted and pointers to monitors keep valid as long as there is at least one remaining reference.

The administrative (B/U) interaction shown in figure 5.93 is now explained in detail. At first, the mutex M1 ① is acquired to ensure that no other administrative interaction is executed concurrently and that the interaction starts only after all sending activities of the interaction patterns of the service requestor are completed ❶. While holding the mutex M2 ②, one can iterate through the *list of monitors* to access the dynamically generated monitor instances of the currently active service related interactions. With a *disconnect*, for example, one has to appropriately set the state automatons of the service related interactions to notify the pending interactions about the connection related state changes. In case of a *disconnect*, for example, these have to know that the expected response cannot be received anymore. The next steps are to first call the *prepare* method of the monitor of the administrative (B/U) interaction, to then send the request message ③ and to finally invoke the *wait* ④.

As explained below, it is mandatory that one neither holds the mutex M2 nor the lock of the monitor of the administrative interaction when performing the *send*. Even if the monitor lock is released for the *send*, no administrative interaction can be invoked concurrently so that no other administrative interaction than the currently executed one accesses the monitor. Thus, although the monitor is accessible, its state does not get messed up by concurrent administrative interactions.

At the service provider, the upcall performs some work and acquires the mutex M10 ⑤ as soon as either the list of connected service requestors needs to be accessed or a response is to be sent ⑥. The *list of monitors* again contains the dynamically generated monitor instances of the service related interactions. With a *disconnect*, for example, one needs to iterate through the *list of monitors* to notify affected service related interactions by appropriately setting their state automatons. In case of the *query* communication pattern, for example, the not yet answered requests of a service requestor, that gets disconnected, need not to be processed anymore. In contrast to the service requestor part, even the monitor lock can be hold while performing the *send* ⑥. Again, the reason why this holds true gets clear with the subsequent explanations on the locking strategy.

At the service requestor, the upcall for the response ⑧ accesses the monitor belonging to the interaction pattern. If the identifier of the received response matches the expected one, the blocked administrative (B/U) interaction resumes processing ⑨. The upcall does not need to acquire the mutex M1 since it leaves the modification of any items that are protected by the mutex M1 to the resumed thread. Of course, the upcall ⑧ can acquire the mutex M2. The resumed method can acquire the mutex M2 only after it released the monitor lock since one always has to take into account the locking order.

The ordering of the mutexes is fixed to prevent from deadlocks and one always has to acquire them in the appropriate order. At the service requestor, that order is M1 first, then M2 and finally a monitor lock. Besides the mutex M1, no locks must be hold while performing a *send* or while awaiting a response. The monitor locks are automatically released when invoking the *wait*. Thus, before acquiring a monitor lock that is automatically released by a *wait*, one always has to release the mutex M2 first. At the service provider, the locks are ordered in the same way that is one first has to acquire the mutex M10 and then a monitor lock. In contrast to the service requestor, both the mutex M10 and a monitor lock can be hold while performing a *send*. In contrast to the mutex M1, however, the mutex M10 must never be hold while awaiting a response as is explained now.

In principle, the availability of the mutex M2 never depends on the availability of any communication activities. It is never hold while awaiting a response or while waiting until a message can be sent. Since the mutex M1 at the service requestor is never acquired by any upcall ❷, all upcalls at the service requestor can always be completed even if the mutex M1 is hold due to an administrative (B/U) interaction that is awaiting its response. Thus, all sending activities from the service provider to the service requestor ⑥ can always be completed independently of the interaction they belong to and thus, sooner or later, the mutex M10 gets released. As soon as the mutex M10 is released, the upcalls at the service provider ❸, that require the mutex M10, can proceed. Thus, the upcall of the administrative (B/U) interaction also gets through and invokes the *send* ⑥ that completes the interaction ⑨. Even if the upcalls at the service provider ❸ get temporarily blocked on the mutex M10, that causes no deadlocks since the release of the mutex M10 never depends on the capacity of the service provider to process further upcalls.

Administrative (B/U) interactions always have the client part (B) at the service requestor of the communication pattern and the server part (U) at the service provider. The *connection oriented split protocol* never requires an administrative (B/U) interaction from a service provider to a service requestor. That is the crucial factor why the administrative (B/U) interaction can never cause a deadlock. If an administrative (B/U) interaction was directed from the service provider to a service requestor, the following deadlock situation would occur. A service requestor invokes an administrative (B/U) interaction and holds and blocks the mutex M1 until the corresponding response arrives. Since the mutex M1 is locked, no other message can leave the service requestor meanwhile. At the same time, the service provider might invoke an administrative (B/U) interaction, locks the mutex M10 and waits until its response arrives. Since it holds the mutex M10, no other message can leave the service provider meanwhile. That, of course, results in a deadlock since neither the service requestor nor the service provider can send the response that is needed to release the locked mutexes.

The Administrative (C/U) Interaction Of course, an administrative (C/U) interaction behaves exactly like the request part of the administrative (B/U) interaction. However, an administrative (C/U) interaction does not await an answer. Thus, neither the mutex M1 nor the mutex M10 needs to be hold beyond the actual *send*. That is the reason why administrative (C/U) interactions can be used in both directions, from the service requestor to the service provider and vice versa.

Figure 5.94 summarizes the behavior of the administrative (C/U) interaction for both directions. At the service requestor, again all upcalls ③ get processed and thus, the mutex M10 at the service provider always gets released. Therefore, the upcalls ④ at the service provider can be processed as well and the messages of both interactions, ① and ②, get processed.

At the service provider, the upcall of an administrative (C/U) interaction can access any mutex protected entities since it can acquire the mutex M10 and since it can access the monitors. At the service requestor, the upcall is not allowed to access the mutex M1 and in contrast to an administrative (B/U) interaction, there is no pending method invocation that can perform the desired processing after getting resumed. Thus, an administrative (C/U) interaction from the service provider to the service requestor always requires a separate thread that is allowed to acquire the mutex M1. In case the mutex M1 is hold by an administrative (B/U) interaction, it gets released only after the response of

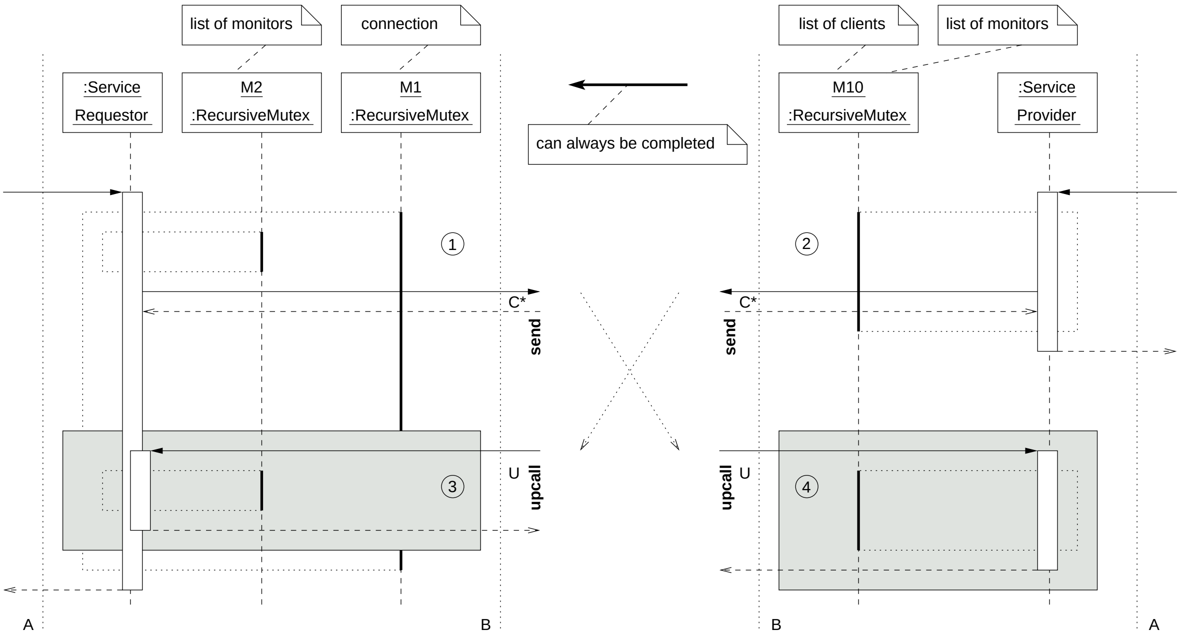


Figure 5.94: *Administrative (C/U) interactions into both directions (*delivered *policy).*

the administrative (B/U) interaction arrived. That requires that all messages prior to the response are also accepted by the service requestor. In principle, there can be any number of administrative (C/U) interactions prior to the response that releases the mutex M1. Thus, one option is to use a single thread and a buffer with unlimited capacity to hold all the upcalls to be processed. Another option is to provide an unlimited number of threads so that each administrative (C/U) interaction is forwarded to its own thread.

The *connection oriented split protocol* requires an administrative (C/U) interaction from the service provider to the service requestor only in combination with the *server initiated disconnect* procedure. As described in section 5.6.6.10, a separate thread and a buffer can be provided very efficiently since all service requestors of a component can share it. Due to the role of that administrative (C/U) interaction, the buffer size is not a critical issue. Thus, not being allowed to access the mutex M1 from inside the upcall at the service requestor does – even with an administrative interaction – not result in a limitation of the *connection oriented split protocol*.

The Service Related Interactions Service related interactions and administrative interactions differ with respect to their level of concurrency. With administrative interactions, one has to make sure that always only one administrative interaction is active. In contrast thereto, service related interactions need to be interleavable arbitrarily to support the interleaved use of a service. Thus, with service related interactions, neither the mutex M1 nor the mutex M10 is hold beyond the actual *send*. In particular, none of the mutexes is hold until a response arrived which is the reason why the (C⋆/U) interactions of a service related interaction get not coupled by mutexes as it is the case for administrative interactions. Service related interactions await a response by blocking on the appropriate monitor but without holding any mutex that protects the current administrative state from being changed. It is the responsibility of the administrative interactions to make sure that all affected service related interactions are properly notified in case of relevant state changes. The locking mechanisms in case of service related interactions are illustrated by means of the (D/V) interaction pattern since one can easily grasp the behavior of the other interaction patterns once the service related (D/V) interaction pattern is made clear.

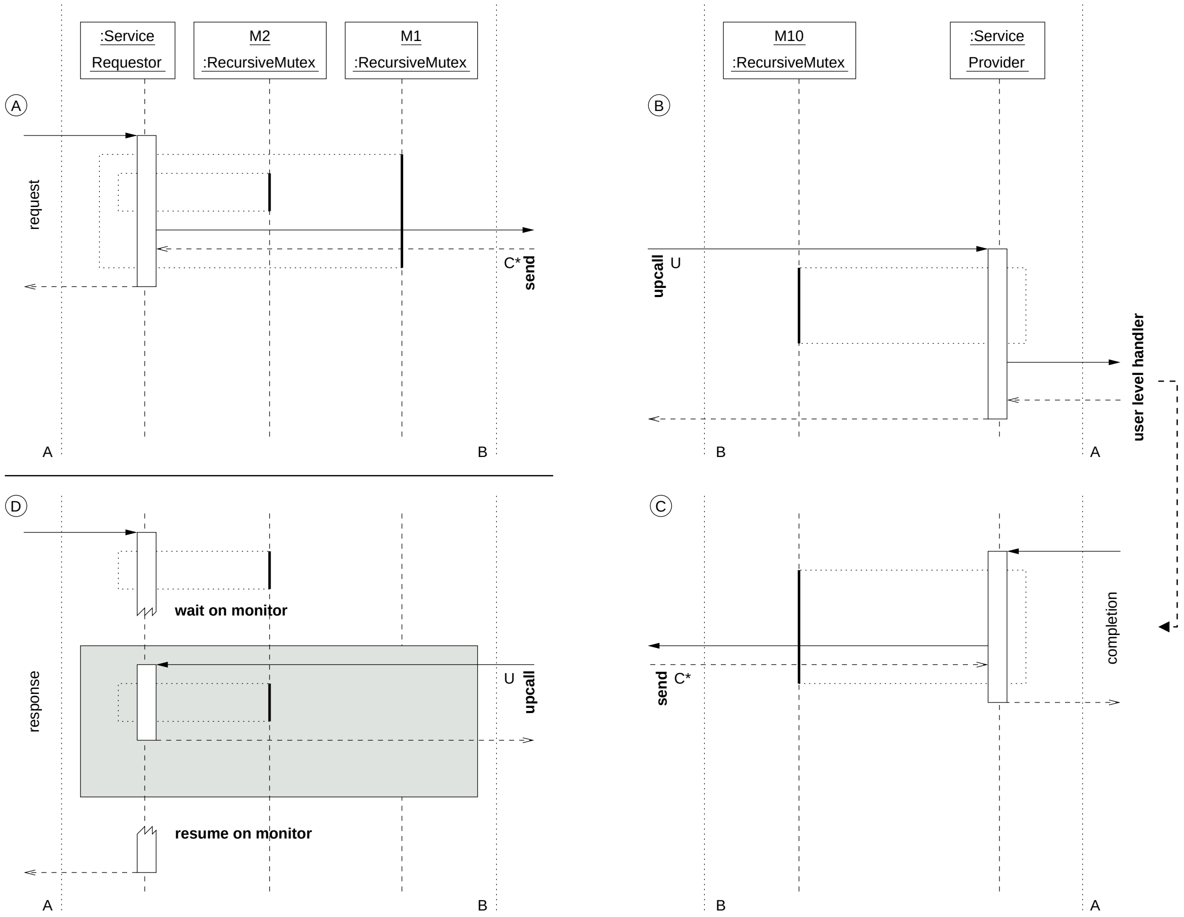


*Figure 5.95: Service related interactions from the service requestor to the service provider (*delivered *policy).*

Figure 5.95 illustrates the service related (D/V) interaction from the service requestor to the service provider. A and B form the first (C*/U) interaction and C and D the second one. A is decoupled from B and C is decoupled from D due to the assumed *delivered* policy. A passive handler uses the upcalling thread to process the handler and we assume that this thread is the one that performs all upcalls. Even now, the upcall of B can invoke C without causing a deadlock since the mutex M10 becomes available independently of the processing of further upcalls. That holds true since the upcall of D always gets completed. Thus, C gets completed as well and then also B which allows the service provider to process further upcalls.

A passive handler is not allowed to access any resources whose availability depends on the upcalling thread. Often, however, one only needs to read a value or to perform a short calculation to generate the answer and the needed resources are available independently of the other responsibilities of the upcalling thread. In those cases, an active handler would be overkill. Since C can be completed independently of the availability of the upcalling thread, a passive handler is allowed to invoke the *completion* method of C to directly answer a request if the above described situation holds true.

Using a passive handler with B results in a structure similar to the administrative (B/U) interaction pattern from the service requestor to the service provider. The major difference is that the mutex M10 is released between B and C. Thus, there can be further interactions that get executed between B and C and one thus has to consider the implications of interleaved interactions.

It is important to recognize that further interactions that are interleaved with the service related (D/V) interaction pattern never cause a problem. All interactions from the service provider to the service requestor always get completed and thus always make way for the interaction from C to D. Lets now consider interleaved interactions from the service requestor to the service provider that are invoked after A. These get delayed in case that the upcall of B is not yet available. However, that does not have any effects on the interactions from the service provider to the service requestor and sooner or later, the upcall at B can proceed. Lets now assume that the service related (D/V) interaction uses an active handler at B and lets further assume, that the upcall of B is available so that further interleaved interactions get executed after B. Now, either C is already running and the upcall of the interleaved interaction has to wait until the mutex M10 gets available which happens in any case or the upcall gets the mutex M10 first so that C has to wait until the interleaved interaction is completed. The interleaved interaction either performs a *send* to the service requestor or invokes a handler. In the first case, the mutex M10 gets released after the *send* that can always be completed. In the latter case, the mutex M10 gets released since it is never hold when invoking a handler. Thus, even an overloaded handler does not block the mutex M10. As consequence, interleaved interactions never interfere even if they all need to access the mutex M10.

Figure 5.96 illustrates the service related (D/V) interaction from the service provider to the service requestor. However, it has to be noticed that a service related (D/V) interaction is normally not directed from the service provider to the service requestor since that would invert the roles of the server and the client part of a communication pattern. Nevertheless, they could still be used with the chosen locking strategy.

In contrast to the administrative (B/U) interaction, two-way interactions from the service provider to the service requestor are allowed with service related interactions since the mutex M10 is always released before the response is awaited. Thus, upcalls at the service provider, that require the mutex M10, can get through even while the service provider is awaiting the response of the service related two-way interaction. Since the mutex M10 is not hold, messages can leave the service provider meanwhile. Thus, even if the service requestor holds the mutex M1 due to a not yet completed administrative (B/U) interaction from the service requestor to the service provider, sooner or later the service requestor receives the message that completes the administrative (B/U) interaction. Thereby, the mutex M1 gets released so that the response of the service related two-way interaction can leave the service requestor. Finally, both interactions are completed.

A precondition for resolving interactions is that service requestors always accept the messages that are addressed to them. This rule is the key towards sorting out all the nested and concurrently active interactions inside a communication pattern. An upcall at the service requestor must never acquire the mutex M1 since the mutex M1 gets available only as long as further upcalls can be processed. Again, the mutex M1 might be hold by an administrative (B/U) interaction that is directed from the service requestor to the service provider. Its response can be accepted only if the thread to process upcalls is available. Furthermore, the handler of L must never induce a tailback in case it invokes M and even on top of a *delivered* policy, it is not sufficient to just have an active handler. Again, one either requires a handler that provides a single thread and a buffer of unlimited size or a handler that provides an unlimited number of threads. Otherwise, the handler could not accept further requests addressed to it. However, these have to be accepted in any case to ensure that the mutex M1 gets released. Only then, the completion method in M can proceed so that the handler can proceed as well.

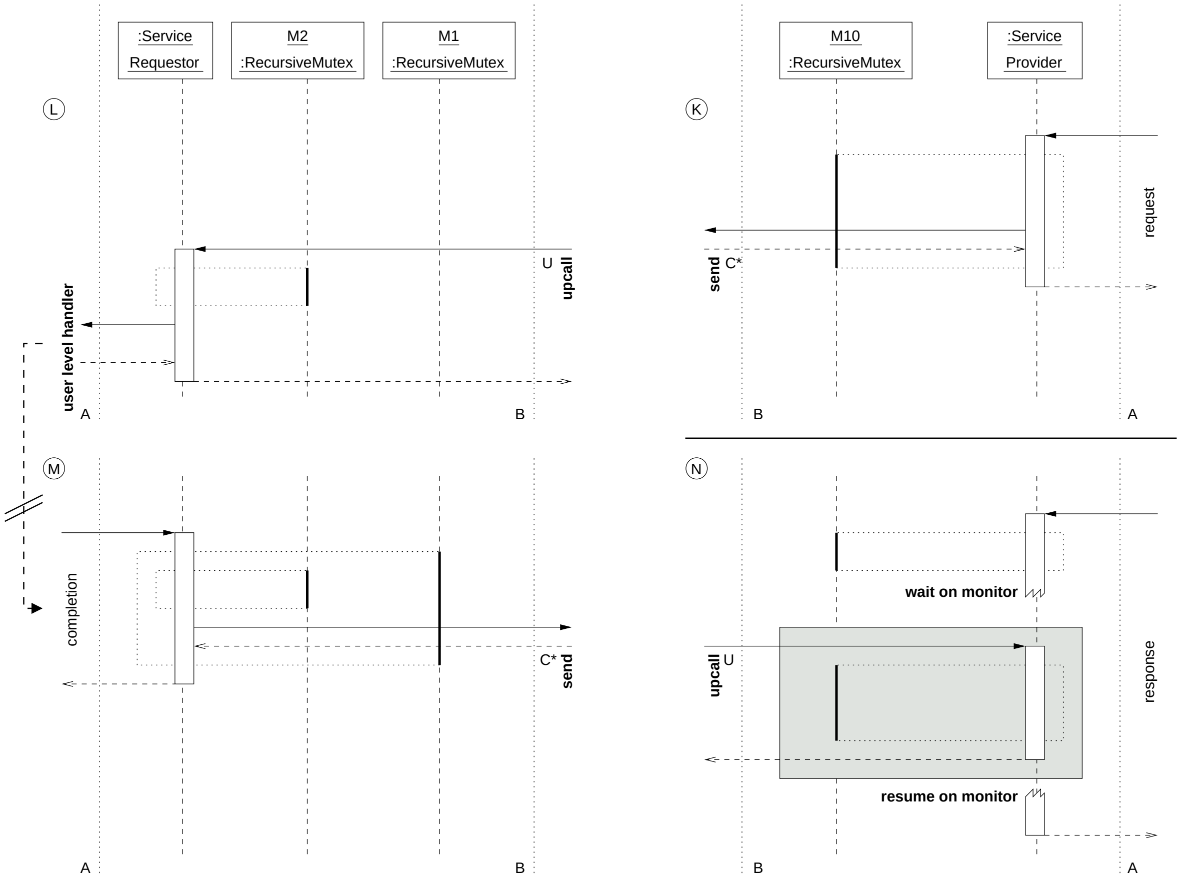


*Figure 5.96: Service related interactions from the service provider to the service requestor (*delivered *policy).*

Thus, in contrast to C, M must never be invoked by a passive handler of L and even only the above two alternatives of an active handler are sufficient.

Of course, the same rules apply for a service-related one-way interaction from the service provider to the service requestor. If it accesses only resources whose availability does not depend on any communication activities, a passive handler is sufficient and a tailback does not cause any troubles since all requests get worked off one after the other. The upcall of L must not invoke a user level handler whose completion depends on resources that get available only in case the upcalling thread can accomplish its further tasks. However, as soon as one accesses any resources whose availability depends on a communication activity, one either needs an active handler with an unlimited number of threads or an active handler with a single thread but with a buffer of unlimited size.

At the service requestor, the upcall of a service related interaction must never acquire the mutex M1 but it also never needs to since it never needs to read states protected by the mutex M1. A service requestor gets service related messages from the service provider only in case it invited the service provider to do so. The service requestor defines which kind of services it calls for. Thus, a service related interaction from the service provider to the service requestor is initiated at the service provider only as long as the service requestor allows the service provider to do so. Thus, these upcalls at the service requestor are always only invoked by messages that are conformant to the states that are protected by the mutex M1 (like the connection state, for example). Thus, there is no need for the service related upcalls to acquire the mutex M1. In case the upcall belongs to a two-way service related interaction, the handler is an active handler to be conformant to the rules for handlers at service requestors. Of course, that thread is allowed to acquire the mutex M1 so that one can send the answer or check whether a response is still needed. In case one now experiences a state change such that the two-way interaction cannot be completed, all pending methods at the service provider part have already been released by the administrative interaction so that discarding the response causes no problems.

Further Remarks on Service Related Interactions Even though service related two-way interactions from the service provider to the service requestor are not a problem with respect to the locking strategies, they are not only avoided due to the assigned roles of the client and the server part of a communication pattern but also to simplify the rules that apply to the user level handlers at service providers. In case of two-way interactions from the service provider to the service requestor, one could end up in the setting shown in figure 5.97. That would require handlers with either unlimited buffer capacity or unlimited number of threads even at the service provider.

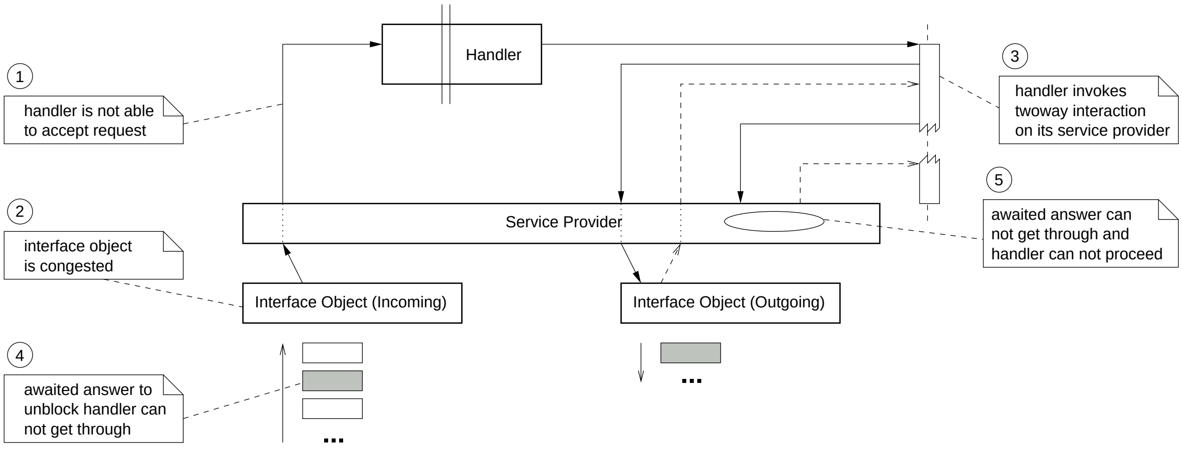

Handler
1
handler is not able to accept request
2
interface object is congested
3
handler invokes twoway interaction on its service provider
4
awaited answer to unblock handler can not get through
5
awaited answer can not get through and handler can not proceed
Service Provider
Interface Object (Incoming)
Interface Object (Outgoing)
...
...

Figure 5.97: Potential deadlock in case of two-way interactions from the service provider to the service requestor with active handlers.

Lets assume that the handler at the service provider is not able to accept any further requests and thus produces a tailback. That prevents further messages addressed to that service provider from getting through the interface object for incoming messages. In case there would be a two-way interaction from the service provider to one of its service requestors, the active handler could invoke that and await the response. Due to the tailback caused by the overload of the active handler, that response could not get through the interface object and thus, the congestion could not be resolved. It is important to note that this kind of deadlock occurs only if the handler does not have unlimited capacity (either with respect to buffer space or the number of threads) and if the handler invokes a two-way interaction at its service provider and if it awaits the response by the handler. The deadlock can be avoided by violating at least one of its preconditions. Even though that can be done easily, the communication patterns avoid that situation by not providing two-way interactions from a service provider to a service requestor.

Of course, the above situation does not occur if the handler sends back an answer, invokes an arbitrary one-way interaction at its service provider or invokes an arbitrary interaction on any other service requestor or service provider.

5.6.6.7 The Locking Strategies Revisited

The locking strategies are now reconsidered on top of a *processed* policy where both parts of an interaction pattern are not decoupled. Administrative interactions do not depend on the decoupling so that the mapping of the *connection oriented split protocol* is simplified significantly.

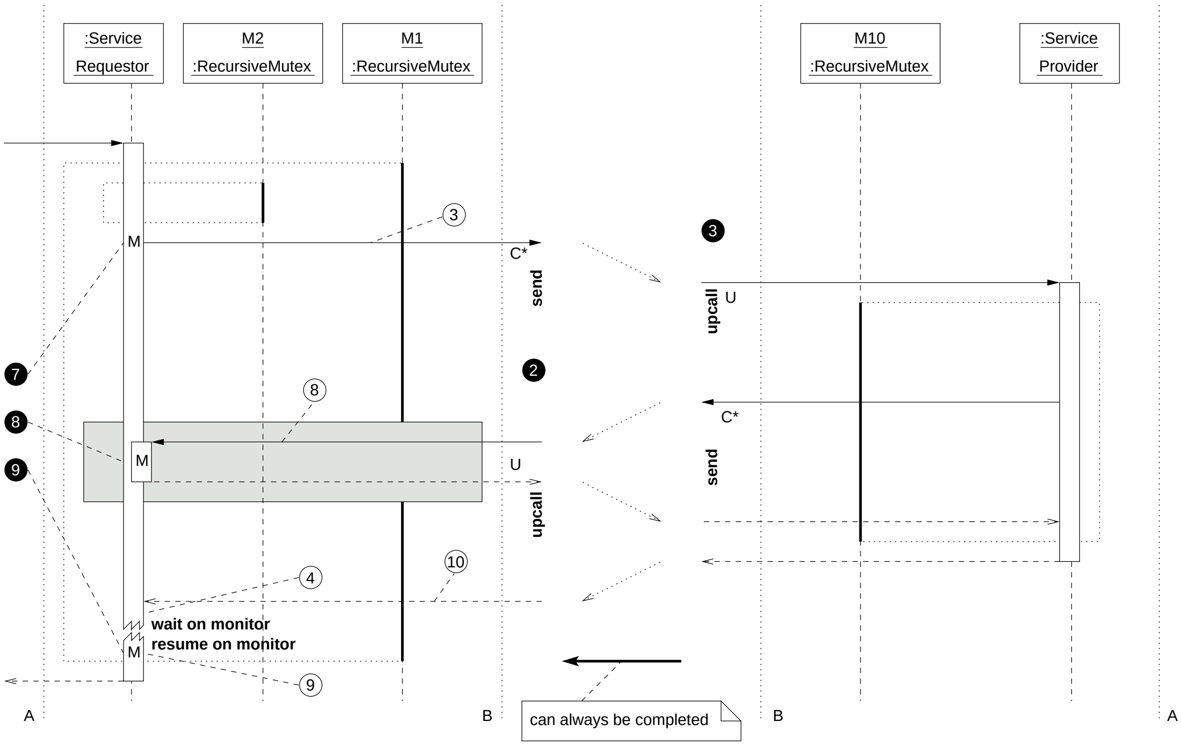


***Figure 5.98:** Administrative (B/U) interaction from the service requestor to the service provider (*processed *policy).*

The Administrative (B/U) Interaction Figure 5.98 shows the administrative (B/U) interaction from the service requestor to the service provider on top of a *processed* policy. The first *M* ❼ again denotes the *prepare* method that is invoked prior to the *send*, the second *M* ❽ the simplified representation of the sequence denoted by *B* in figure 5.91 and the third *M* ❾ the simplified representation of releasing the monitor lock after it is not needed anymore.

Due to the *processed* policy, the *send* ③ from the service requestor to the service provider is completed only after the upcall ⑧ already returned the response. Thus, it is important that the *send* method ③ does not hold the monitor lock while it is invoked. Otherwise, one could not complete the upcall ⑧ that provides the response. The same holds true for the mutex M2. In case the upcall ⑧ needs to access the *list of monitors* that would cause a deadlock. The *wait* ④ returns immediately since the awaited response is already available ⑨. Again, all upcalls ❷ can always be completed independently of any pending administrative (B/U) interaction. That holds true since the upcalls belong to different interface objects. The upcall ⑧ is that of the interface object ② in figure 5.80 and the upcall ⑩ that of the interface object ①. Thus, the mutex M10 always gets released and all upcalls ❸ at the service provider and further concurrent interactions can sort out each other without a deadlock.

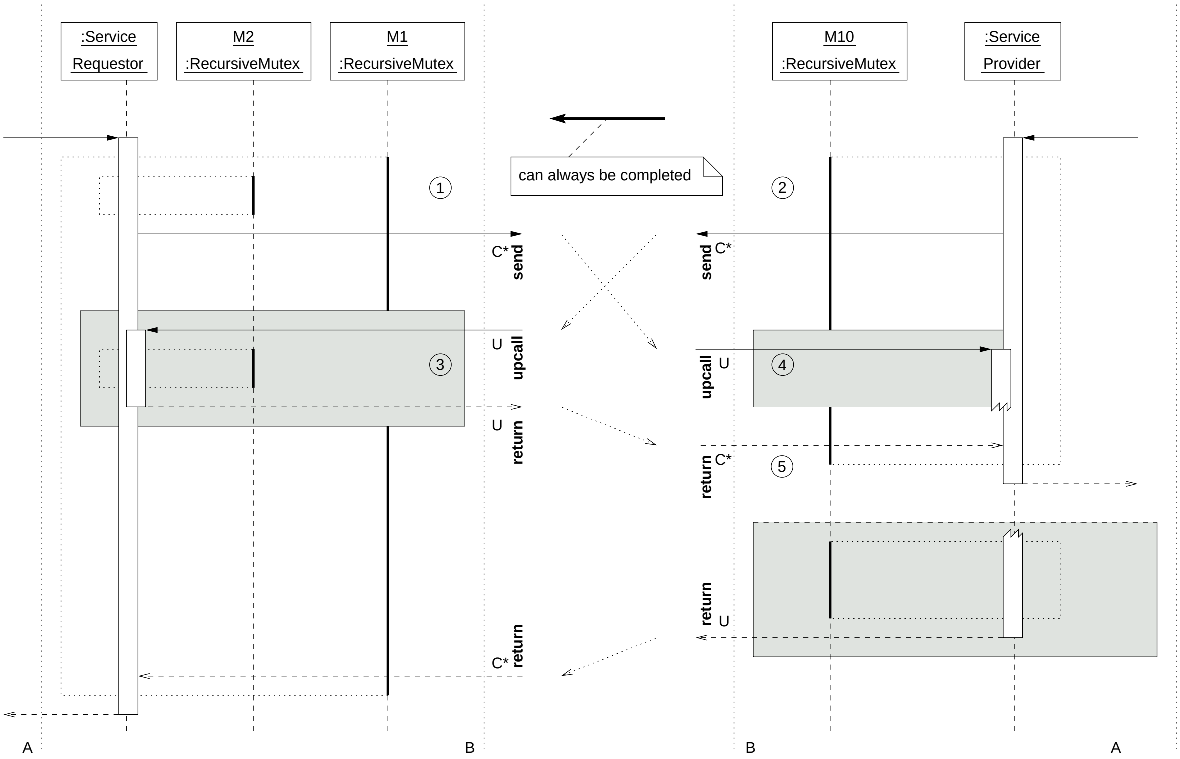


*Figure 5.99: Administrative (C/U) interactions into both directions (*processed *policy).*

The Administrative (C/U) Interaction Figure 5.99 shows two administrative (C/U) interactions on top of a *processed* policy. The upcall ④ gets blocked on the mutex M10 as long as ② is active. The upcall ③ can always be completed independently of how long the mutex M1 is locked since the upcalls at the service requestor never require the mutex M1. Thus, ② gets completed as well and the mutex M10 gets released. Now, the upcall ④ gets resumed and, finally, ① gets completed. It is important to note that even if the upcalling thread ④ at the service provider gets blocked that does not prevent the *send* ② from getting completed. Most important, the blocked upcall ④ never locks up the *return* path ⑤. That holds true since the upcalls belong to different interface objects. The upcall ④ is that of the interface object ④ in figure 5.80 and the upcall ⑤ that of the interface object ③.

The Service Related Interactions A service related (D/V) interaction from the service requestor to the service provider behaves on top of a *processed* policy exactly like the administrative (B/U) interaction. Again, no deadlocks can occur but the service requestor and the service provider are decoupled only if the upcall of B uses an active handler. In all other cases, the asynchronous client part characteristic becomes obsolete.

Lets now consider a service related (D/V) interaction from the service provider to the service requestor. In principle, the mutex M10 must always be released before the response is awaited. Of course, that rule must also be observed on top of a *processed* policy. Thus, the *send* of K must be completed before the upcall N is invoked. Otherwise, one ends up in the same deadlock situation that can arise with administrative (B/U) interactions from the service provider to the service requestor. An active handler is needed anyway to decouple L and M. Of course, that handler also decouples K and

N and is thus sufficient to prevent from deadlocks. As before, the handler of L must never induce a tailback in case it invokes M.

5.6.6.8 Coaction of Interaction Patterns Inside a Communication Pattern

Lets now solely consider the interaction patterns of a single communication pattern. At the service provider, one can use the thread that handles all the upcalls to invoke the completion method that returns a response to the service requestor. At the service requestor, a handler must always be able to accept messages. As outlined above, one either needs an unlimited buffer size or an unlimited number of threads in case one invokes another interaction pattern on the same communication pattern at the service requestor. In all other cases, even a passive handler is sufficient.

The locking mechanism is free of deadlocks as long as no upcall at the service requestor acquires the mutex M1. At the service requestor, all messages must always be sent from outside the mutex M2 and from outside the scope of a monitor. At the service provider, the mutex M10 must never be hold while waiting for a response of a service related two-way interaction. Of course, one always has to follow the mutex locking order.

At the service requestor, the mutex M1 needs to be hold while awaiting a response since otherwise administrative interactions can interfere with each other. Thus, one cannot merge the responsibilities of the mutexes M1 and M2 since upcalls have to be able to search through the *list of monitors*. One is not allowed to hold the mutex M2 nor the monitor lock while performing a *send* since in case of a *processed* policy, the upcall is invoked before the *send* returns. Since there are no administrative (B/U) interactions from the service provider to the service requestor, there is no need to hold the mutex M10 while waiting for a response. Since upcalls at the service requestor are anyway decoupled in case they invoke another interaction back to the service provider, even on top of a processed policy, the mutex M10 always gets released after the *send* of the service provider. Thus, at the service provider, the mutex M10 can also protect the *list of monitors*.

A communication pattern can host any number of interaction patterns. Independently of the interaction pattern type, all sending activities from the service provider to the service requestor can always be completed since the upcall at the service requestor never needs to acquire the mutex M1 and since the coordinating monitors can always be accessed. Client parts of an administrative (B/U) interaction are always located at the service requestor. Thus, the service provider never needs to hold the mutex M10 while awaiting a response. As consequence, the release of the mutex M10 does not depend on the arrival of a certain message and is thus also independent of the ability of the service provider to process an upcall from the communication system. The mutex M10 always gets released even if the thread is blocked that handles the upcalls at the service provider. Thus, the upcall that blocked on the mutex M10 gets resumed as well and further upcalls at the service provider also get processed. Thus, sooner or later, even an administrative (B/U) interaction from the service requestor to the service provider is completed so that the mutex M1 at the service requestor also always gets released.

The property that the service requestor is always able to process the messages addressed to it is the key feature why interactions between both parts of a communication pattern always sort out each other without deadlocks. Another important feature of the locking mechanism is that all upcalls can always be serialized without introducing deadlocks. At the service requestor, that is obvious since all upcalls always get through. At the service provider, that holds true since the availability of the mutex M10 does not depend on processing other upcalls of that service provider. The mutex M10 gets available as soon as the *send* operation to the service requestor is completed and since that can always be completed, the processing of upcalls does not get stuck. Thus, at the service provider, the serialization does also not cause a deadlock.

Furthermore, at the service provider, upcalls from the communication system need not to be decoupled before invoking the communication patterns. Since the administrative interactions even work on top of a *processed* policy, one solely has to consider the service related interactions. Since these all invoke a user level handler, one can delegate the decoupling task to the user level handlers by providing appropriate handlers that ensure the decoupling without requiring any further user level interventions.

At the service requestor, the upcalls from the communication system do also not depend on a decoupling mechanism below or inside the communication patterns. As shown in sections 5.6.5.1 and 5.6.6.1, a service related upcall at a service requestor either belongs to an interaction pattern where the actual processing must be invoked by the user or it belongs to a two-way interaction pattern that is always directed from the service requestor to the service provider. In the latter case, the upcall at the service requestor solely returns the answer. In all cases, there is no further processing invoked by the upcall at the service requestor so that there can be no client part characteristics of an interaction pattern be violated in case of not decoupling the service related upcall at the service requestor.

The *event* pattern is the only pattern where the upcall at the service requestor can invoke the further processing. The further processing, however, is done by means of a user level handler so that appropriate handler types are again sufficient to achieve the desired decoupling. All other activities that are performed inside the service requestor prior to invoking the user level handler do not depend on any kind of decoupling so that, again, a handler based decoupling model is sufficient.

As described in section 5.6.6.1, the *query* pattern, the *push* patterns and the *event* pattern are always able to accept answers, updates and event firings, respectively. In all cases, the required structures to accept the messages are available since the message is either expected as in case of the *query* pattern or it overwrites a buffer as it is the case with the *push* and the *event* patterns. Thus, that part of the service related interactions that is handled inside the communication patterns, is also not critical since it does not require unlimited buffer space to keep the communication pattern alive. In case of the *query* pattern, another request can only be invoked if both the client and the server are able to set up the required administrative structures. Thus, if the *query* pattern runs out of memory, one already cannot invoke further requests which would produce further responses. The *push* and *event* patterns do not depend on the user to fetch any received data due to their overwriting behavior.

As consequence of both the serialization and the decoupling that is performed earliest inside the user level handlers but still beneath the user level implementation, one does not already need elaborate threading models at the level of the upcalls so that one can provide various threading models by means of different handler types. This leaves it then to the user to select the appropriate processing models and does not introduce numerous threads or buffers above the communication system and beneath the user level. As additional remark, not depending on the decoupling for the administrative interactions is equivalent to ignoring the *oneway* statements for the administrative interactions in tables 5.45 and 5.46. One can even merge the request/response pairs of an administrative interaction into a single synchronous two-way remote method with the content of the response part as *out* arguments. Since the administrative interactions require neither separate threads nor a decoupling layer, they can be implemented very resource friendly.

With the communication patterns, the *server initiated disconnect* is the only administrative interaction that needs to invoke another interaction and that thus depends on a decoupling mechanism inside the service requestor part of a communication pattern. The upcall at the service requestor needs to invoke the *disconnect* procedure that, of course, contains an interaction back to the service provider. An upcall at the service requestor, however, must never invoke another interaction by the thread that is responsible for handling the incoming messages. Furthermore, a service requestor must always be able to accept all messages addressed to it. Nevertheless, in case of the *server initiated disconnect*, the decoupling can be achieved very efficiently as described in section 5.6.6.10.

5.6.6.9 Coaction of User Level Handlers and Communication Patterns

The *handlers* of the communication patterns can have a significant impact on the overall behavior of a communication pattern. At service requestors, handlers are critical as soon as further interactions are invoked on the same communication pattern. Thus, one approach is to avoid user level handlers at service requestors. That is the approach taken by the communication patterns with the only exception of the *event* pattern. In particular, there is no handler interface provided at the service requestors of the *push* patterns.

A handler at a service requestor can be passive in case it accesses only such resources that in no way depend on any communication activities. Even if the handler then causes a tailback, the communication pattern can still sort out its interactions since the service requestor still works off one message after the other independently of the congested communication paths. However, invoking another interaction on the service requestor requires an active handler with either unlimited buffer size or with an unlimited number of threads.

Normally, two types of handlers are provided with the *event* pattern, the *passive* and the *active* handler. The *active* handler operates a queue by a single thread and provides buffering for events to compensate for bursts. In case the queue size gets too large, further events are dropped which can be detected by the user. The *passive* handler is operated by the upcalling thread and does not drop any events. It causes delays as soon as the buffers of the communication system are filled up. Thus, it is not advised to use it beyond signalling and forwarding of events. The *passive* handler must not block on any communication dependent resource. Thus, it is neither allowed to access any method at its service requestor nor any method at any other communication pattern.

On top of a communication system with a *processed* policy, the *passive* handler is implemented as single-threaded handler with a fixed buffer size without dropping events. That achieves the desired decoupling transparently to the user without changing the semantics of the handler. The handlers of the *event* pattern do not introduce deadlocks as long as one does not wrongly apply the *passive* handler since the service requestor still appears as unbounded sink. Even a tailback caused by the *passive* handler gets resolved.

The strict rules for handlers at service requestors do not impose any restrictions on the concerned communication patterns. In principle, the service requestor misses data only if the capacity of the active handler either in terms of buffer size or number of threads is exceeded. Since a handler at a service requestor is invoked for data the service requestor asked for, it is the fault of the service requestor that it is not able to perform the processing in time. In particular, a service requestor should never obtrude a service provider its timing.

Of course, in practice, there are rare cases where the buffer size would grow to a remarkable size. In case of the *event* pattern, only a *continuous* activation could overrun the handler by too many firings and the service requestor can always deactivate the corresponding event respectively select the activation parameters more carefully. The same considerations apply to the service requestor parts of the *push* patterns in case of adding a handler based interface.

Service providers also distinguish *passive* and *active* handlers. However, *passive* handlers can even be used to send back responses to connected clients without causing a deadlock since service requestors are always able to accept all incoming messages. Of course, an *active* handler of a service provider is allowed to access further services. However, the type of handler determines what kind of dependencies between components are supported. For example, an active handler with a single thread can not process further requests while waiting for a response to complete the processing of the current item. Thus, a service provider using this type of handler must never be wired such that a circular processing dependency occurs. However, due to the high level abstraction of services compared to fine-grained method access, there are rare cases where circular processing dependencies make up a reasonable configuration. Nevertheless, one can use a handler with a thread-per-client model or even a handler with a thread-per-request model to tackle such settings. In most cases, services are naturally specified such that an active handler with a queue and a single thread is sufficient to cover the reasonable wirings of an application. As soon as the queue reaches a watermark, for example, the *discard* method is called by the service provider to reject requests.

5.6.6.10 The Connection Management Procedures

The administrative interactions of the connection management form the core of the *connection oriented split protocol*. The *connect* and the *disconnect* procedure are invoked from a service requestor and the *server initiated disconnect* procedure is invoked from the service provider. The connection management ensures that connected service requestors and service providers always know when their opponent disappears. It implements an environment in which all interactions can rely on a *delivered* policy even if the underlying communication system provides a *reliable send* policy only. It is sufficient to achieve a *delivered* policy for the *connect* procedure since once a service requestor is connected to a service provider, both inform each other about getting deleted and becoming unreachable. The designators of the messages refer to tables 5.45 and 5.46.

The Connect Procedure The *connect* request R0 and the response A0 form an administrative (B/U) interaction from the service requestor to the service provider. The request R0 provides the address of the service requestor and a *connection identifier*. The address of the service requestor enables the service provider to inform the service requestors connected to it in case it gets destroyed. The connection identifier is generated by the service requestor and it is changed with every connect that is with every invocation of the *connect* procedure. It uniquely identifies each connect procedure within a service requestor. Even if a *connect* fails, one is not allowed to reuse the connection identifier. As described with the *server initiated disconnect* procedure, one could otherwise not identify outdated *server initiated disconnect* messages in all cases. The response A0, that corresponds to a particular request R0, can be identified by the connection identifier so that outdated A0 acknowledgments can easily be ignored. The returned status of the response A0 can be either *accepted* or *rejected*. In case the service requestor receives an *accepted* status, it got connected to the service provider and its address was added to the *list of clients* at the service provider. A *reject* status indicates that the service provider did not accept the *connect* and thus, there is no connection established to the service provider.

Generally, a service provider can disappear after a service requestor got its address from the name service and before it invokes the *connect* procedure. Due to the underlying ($C^\star$/U) interaction, one cannot rely on an appropriate feedback from the communication system in case the service provider does not exist anymore. Thus, the waiting time for the response A0 is limited by a timeout. If no timeout occurs, the outcome is the above described standard *connect* procedure. In case of a timeout, one has to distinguish several scenarios. An overview on the various *connect* procedures is given in figure 5.100.

In the first case, the service provider still exists but the timeout occurred before the response A0 arrived at the service requestor. The service requestor assumes that it could not connect to the service provider. However, sooner or later, the request R0 arrives at the service provider and the service requestor gets added to the list of clients. That is the reason why the service requestor always sends

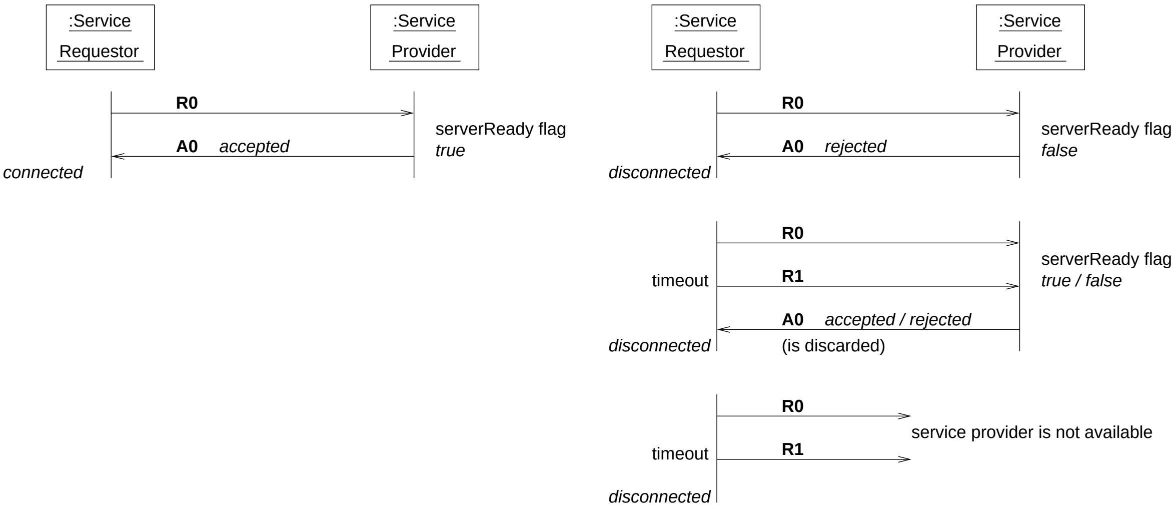


Figure 5.100: *Overview on the* connect *procedure.*

the message R1 in case of a timeout of the response A0. The message R1 removes the newly added service requestor from the *list of clients* at the service provider. Otherwise, the service provider would assume that the concerned service requestor is connected to it whereas the service requestor assumes that the connect failed.

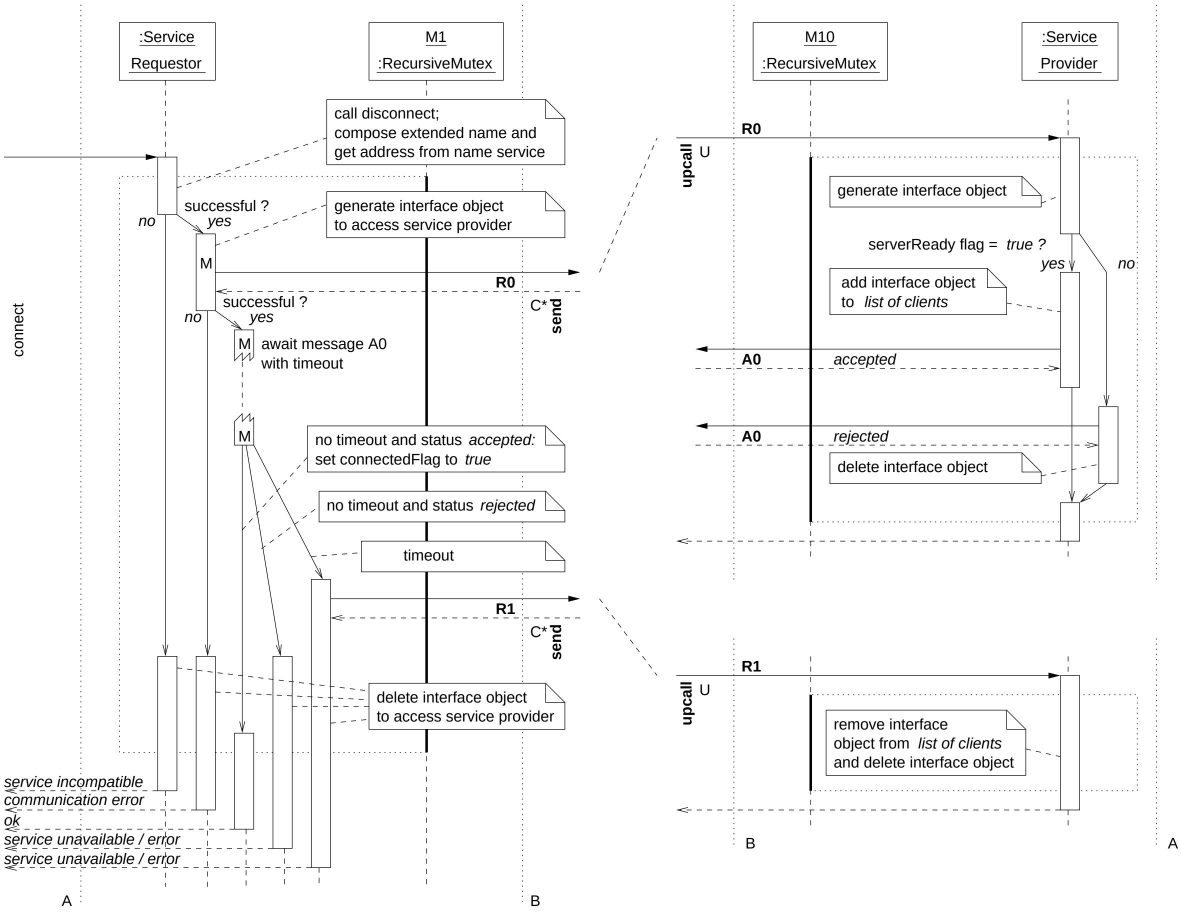
The message R1 forms an administrative (C/U) interaction from the service requestor to the service provider. At the service provider, one has to clearly identify the entry that belongs to the service requestor to be removed. The *client address* is sufficient since each service requestor can be listed only once which holds true since a service requestor can be connected to a service provider only once at a time. Furthermore, due to the kept ordering of messages, there can be no late R1 message at the service provider since a R1 message always arrives before the next R0 message of that service requestor can arrive. Thus, the connection identifier is not needed to identify the proper entry in the *list of clients*. The message R1 must be invoked after a timeout of the R0/A0 interaction occurred but from inside that section that is protected by the mutex M1 of the R0/A0 interaction. It is important that the mutex M1 is not released after the R0/A0 interaction is completed and before the R1 interaction is invoked since otherwise further administrative interactions could interfere with the not yet completed *connect* procedure.

The regular case of a timeout is a service provider that has disappeared. Again, the message R1 is sent by the service requestor but now, it simply gets dropped by the communication system as it was already the case for the message R0.

The timeout value to await the response A0 is not critical. Even with a too small timeout value, one cannot mess up the protocol and a too large value is not problematic since it affects the *connect* procedure only. That is executed rarely compared to the regular interactions performed by a communication pattern and a reduced efficiency of a *connect* in case of a no more existing service provider has no effect on the efficiency of the current interactions once a connection is established.

In principle, the timeout is needed only on top of a *reliable send* policy but it does not disturb on top of other policies. On top of both the *delivered* and the *processed* policy, it actually is obsolete since the *send* already gives the feedback whether the recipient does not exist anymore. With these, one could avoid the timeout if one does not invoke the *wait* that awaits the response A0 in case the *send* was not successful. However, not using a timeout does not allow to discard the contents of buffers so that even on top of a *reliable send* and a *processed* policy, a timeout not only makes sense but also avoids implementational differences.

In contrast to the service provider, there can be late messages at the service requestor. The response A0 can arrive after a timeout occurred and even after the next *connect* is invoked. However, due to the *connection identifier*, it is never mixed up with the awaited response A0.



:Service Requestor
M1 :RecursiveMutex
M10 :RecursiveMutex
:Service Provider
call disconnect; compose extended name and get address from name service
successful ?
no
yes
generate interface object to access service provider
M
R0
successful ?
no
yes
C*
send
connect
M await message A0 with timeout
M
no timeout and status *accepted*: set connectedFlag to *true*
no timeout and status *rejected*
timeout
R1
C*
send
delete interface object to access service provider
service incompatible
communication error
ok
service unavailable / error
service unavailable / error
A
B
R0
upcall
U
generate interface object
serverReady flag = *true ?*
yes
no
add interface object to *list of clients*
A0
accepted
A0
rejected
delete interface object
R1
upcall
U
remove interface object from *list of clients* and delete interface object
B
A

***Figure 5.101:** Details of the* connect *procedure.*

Figure 5.101 shows the details of the involved interactions of the *connect* procedure. After the service requestor obtained the address of the service provider from the name service, it generates the interface object to access the service provider. In figure 5.80, that interface object is labeled by ①. In case the message R0 was sent successfully, the answer A0 is expected with a timeout.

At the service provider, the upcall of R0 first generates the interface object that is labeled by ③ in figure 5.80. It then checks the *server ready* flag. In case the service provider accepts the new *connect*, the interface object is added to the *list of clients* and the message A0 is sent with an *accepted* status. Otherwise, the message A0 is sent with a *rejected* status and the just generated interface object is deleted. It is important to release the mutex M10 only after the upcall R0 is completed and to send the message A0 without releasing the mutex M10 in between. Otherwise, a *server initiated disconnect* might interfere at the service provider.

A service provider accepts a new *connect* only while its *server ready* flag is set to *true*. The *server ready* flag protects the service provider from new connects when it is not yet ready or when it is already in the process of destruction as it is described with the *server initiated disconnect*. Being able to reject further requests is important in connection with destroying a service provider. In that situation, one still has to keep the communication alive to properly complete all interactions such that the service provider can reach a state in which it can be safely shut down without leaving any pending activities at service requestors. In that stage, of course, no further *connect* is accepted. Such a *connect* can result from a service requestor that still got the address of the service provider from the name service but invokes the *connect* procedure only after the service provider started its destruction.

At the service provider, the upcall of R1 simply removes the interface object belonging to the address of the service requestor from the *list of clients* and deletes the interface object. With the communication patterns, all interactions from a service provider to a service requestor require preceding interactions from the service requestor like a *query*, an *activate* or a *subscribe*. As long as the *connect* procedure is not successfully completed at the service requestor, the mutex M1 is not released. Thus, no other interactions from the service requestor to the service provider can be invoked that would require further cleanup activities within the upcall of R1 at the service provider. That is also the reason why it is unproblematic to already add the address of a service requestor to the *list of clients* with the message R0.

With the communication patterns, there are no interactions from a service provider to a service requestor that are invoked by the service provider solely based on the fact that the address of a service requestor is contained in the *list of clients*. Otherwise, the following situation might occur. A service requestor sends the message R0 and gets added to the *list of clients*. However, the service requestor might encounter a timeout with respect to the response A0, sends the message R1 and destroys itself. The service provider invokes an interaction after it received the message R0 and after the service requestor got destroyed but before the message R1 arrived at the service provider. Now, the already invoked interaction should be informed by the upcall of R1 that there can be no answer since one wrongly assumed a valid connection to that service requestor.

In case of the communication patterns, the above described situation cannot occur. Once a *subscribe*, an *activate* or a *query* is received that would require clean up activities, there can be no deferred R1 message since then the *subscribe*, the *activate* and the *query* would have passed a R1 message. Thus, all interactions, that are invoked from the service provider to the service requestor, are only performed on established connections and are already covered by the *delivered* guarantee of the connection management.

The only interaction that is invoked on the *list of clients* and for which the above situation can arise is the *server initiated disconnect*. However, the *server initiated disconnect* does not expect an answer and as described below, a discarded *server initiated disconnect* due to a disappeared service requestor imposes no troubles. A service requestor always either regularly disconnects itself or it sends the message R1 so that the entry in the *list of clients* gets removed for sure.

The Disconnect Procedure The *disconnect* request R2 and the response A2 also form an administrative (B/U) interaction from the service requestor to the service provider. The request R2 provides the address of the service requestor to be disconnected and the response A2 acknowledges the completion of the *disconnect* procedure.

Figure 5.102 shows the details of the involved interactions. The mutex M1 not only ensures that the *disconnect* procedure never interferes with a currently active *send* but it also blocks other interactions from the service requestor while the *disconnect* procedure is active. The first step is

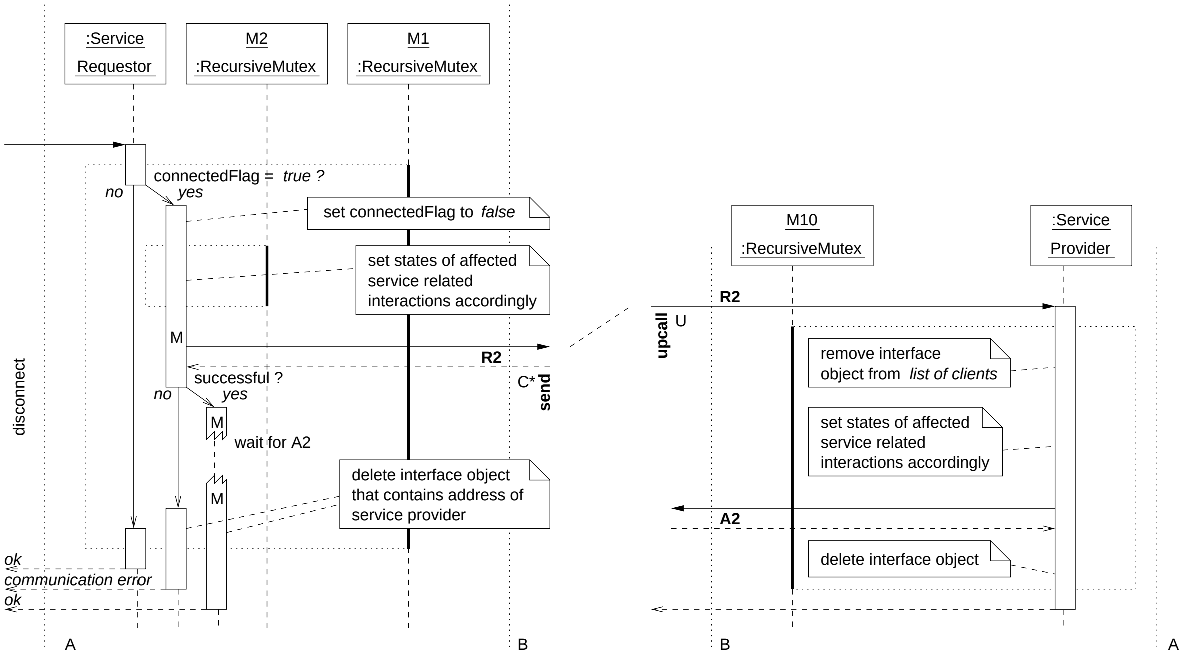

:Service Requestor
M2 :RecursiveMutex
M1 :RecursiveMutex
connectedFlag = true ?
no
yes
set connectedFlag to false
set states of affected service related interactions accordingly
M
R2
successful ?
no
yes
C*
send
M
wait for A2
M
delete interface object that contains address of service provider
disconnect
ok
communication error
ok
A
B
M10 :RecursiveMutex
:Service Provider
R2
upcall
U
remove interface object from list of clients
set states of affected service related interactions accordingly
A2
delete interface object
B
A

***Figure 5.102:** Details of the* disconnect *procedure.*

to set the *connected* flag to *false*. Thus, all further interactions, that depend on a connection to a service provider, are rejected as soon as they can get the mutex M1 after the *disconnect* procedure is completed.

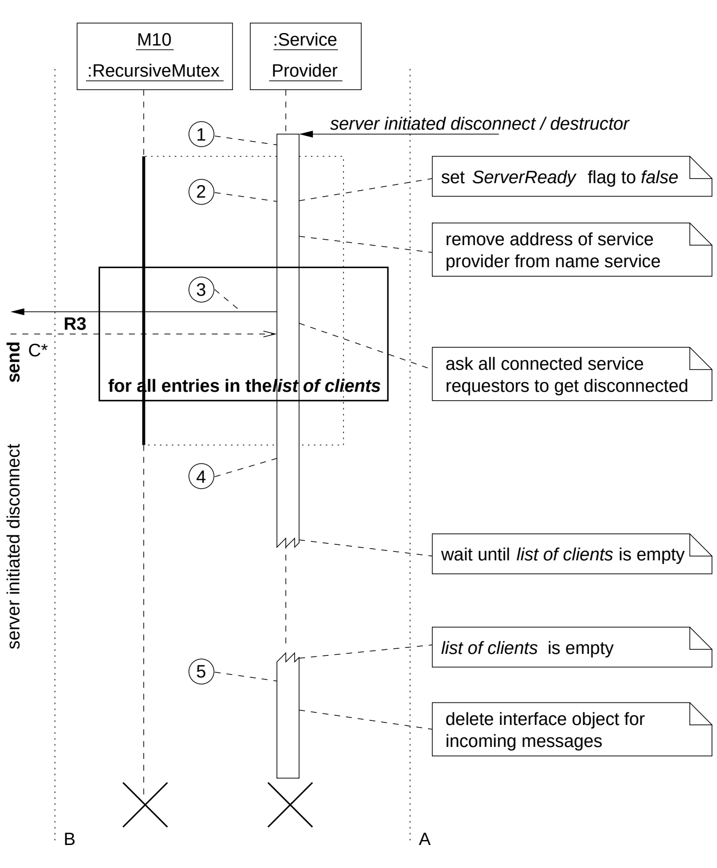
The second step is to iterate through the list of monitors. All pending service related interactions are informed about the *disconnect* by performing the appropriate state transitions on the state automatons of the monitors of the service related interactions not yet completed. As a consequence, blocking calls that await a not yet received response get released properly. From now on, still arriving responses are discarded by the upcall of the service requestor either due to the state of the corresponding state automaton or since the corresponding monitor does not even exist anymore. Responses can still arrive as long as the *disconnect* is not completed at the service provider.

The next step is to invoke the R2/A2 interaction. At the service provider, the interface object of the corresponding service requestor gets removed from the *list of clients* and all affected service related interactions belonging to that service requestor are informed properly by setting their state automatons to the appropriate state. After the response A2 is sent, the interface object, that corresponds to the just disconnected service requestor is deleted. The deleted interface object is one of those denoted by ③ in figure 5.80.

At the service provider, the mutex M10 is never released between getting the address of a connected service requestor and sending the message to that address. Otherwise, an R2 upcall could be executed in between and the address might get invalid if the service requestor destroys itself once it got the message A2. Without releasing the mutex M10, the address either remains valid until the *send* is performed or it is not anymore contained in the *list of clients*. In the latter case, one cannot invoke an interaction since one does not get the address.

The service requestor always awaits the response A2. Since the response A2 always gets consumed before the next *disconnect* interaction can be invoked, there can be no outdated A2 responses so that no identifier is needed with them. In contrast to the *connect* procedure, no timeouts are needed since a service provider never disappears while there are still clients connected to it. Thus, it always completes the *disconnect* procedures which includes sending the response A2.

The Server Initiated Disconnect Procedure The *server initiated disconnect* procedure is an administrative (C/U) interaction from the service provider to the service requestor. By means of the *server initiated disconnect* message, the service provider asks all its connected service requestors to disconnect themselves from the service provider by using the regular *disconnect* procedure. The *server initiated disconnect* procedure is invoked only in case the service provider gets destroyed. Thus, it is invoked from inside the destructor of the service provider.


M10
:RecursiveMutex
:Service
Provider
server initiated disconnect / destructor
1
2
3
4
5
set *ServerReady* flag to *false*
remove address of service provider from name service
R3
C*
send
for all entries in the *list of clients*
ask all connected service requestors to get disconnected
server initiated disconnect
wait until *list of clients* is empty
list of clients is empty
delete interface object for incoming messages
B
A

Flag is never set back to *true*
From now on, all connects are rejected

No service requestor can get the address anymore. However, there might be some service requestors that still got the address but that have not yet invoked the connect procedure

Ignore a failure caused by an invalid service requestor address since then a R1 message removes the address from the *list of clients*

Releasing the mutex M10 allows the pending upcalls to proceed. Due to the server ready flag, all new connects are already rejected so that there are no further requestors added to the *list of clients*

Service requestor gets removed from *list of clients* by either a disconnect procedure or a R1 message

As soon as the *list of clients* is empty, there can be only left over *connects.* Not answering them simply results in a timeout so that it is safe now to destroy the interface object.

One has to ensure that the interface object is not destroyed while there is an active upcall from the interface object into the service provider.

Figure 5.103: *The service provider part of the* server initiated disconnect *procedure.*

Figure 5.103 illustrates the service provider part of the *server initiated disconnect*. First of all, the mutex M10 is acquired. Setting the *server ready* flag to *false* rejects all further *connects* after the mutex M10 is released. Furthermore, the address of the service provider is removed from the name service so that service requestors cannot find the service provider anymore. However, there might still be service requestors that already got the address but that have not yet invoked a *connect* procedure. Finally, all currently listed service requestors are asked to disconnect themselves from the service provider. Since the mutex M10 is not released before the message R3 was sent to all listed service requestors, there can be no upcall in between that modifies the *list of clients*. After having sent the R3 messages, the mutex M10 has to be released since otherwise the upcalls of the *disconnect* procedures cannot be processed. The service provider waits until the *list of clients* is empty. Due to the *server ready* flag, no further *connects* are accepted and since all connected service requestors are asked to disconnect themselves, sooner or later the *list of clients* gets empty. From now on, there are no more clients connected to the service provider. Therefore, the only messages addressed to that service provider can be delayed *connects* that result from service requestors that got the address before it was removed from the name service. Those messages, however, can be ignored since the corresponding service requestor simply experiences a timeout. Thus, the interface object for incoming messages, that is labeled by ④ in figure 5.80, can be deleted. Once that interface object is deleted, one can destroy the service provider itself.

A service provider must always be prepared to receive a delayed *connect* as long as its interface object for incoming messages is accessible. Although the R0 messages do not have to be answered, one has to make sure that the interface object is not deleted while an upcall due to a delayed R0 message is performed from the interface object to the service provider. In case of *CORBA*, for example, it is sufficient to destroy the servant object. The already delivered requests respectively the currently executed methods are still completed but further requests are already rejected. As soon as the servant object got deleted, it is also safe to delete the interface object since there can be no more active upcalls. With an approach that is, for example, based on *TCP sockets* or *mailboxes*, one first shuts down the socket and the mailbox, respectively. The next step is to shut down the thread that is responsible for reading the messages and performing the upcalls. It can be shut down safely as soon as it resides in the interface object. Now, it is safe to destroy the interface object. All messages not yet processed that are stored in any buffers can be ignored since these can only be R0 messages.

Figure 5.104 illustrates the service requestor part of the *server initiated disconnect* procedure. The upcall at the service requestor just has to invoke the regular *disconnect* of the service requestor. Since an upcall at the service requestor is not allowed to invoke an interaction that requires the mutex M1 and since the ability of a service requestor to process further upcalls must never be blocked, the upcall of the message R3 depends on a decoupling inside the service requestor. In particular, the chosen decoupling mechanism has to be able to accept an unlimited number of *server initiated disconnect* requests. However, the R3 messages of all service requestors of a component can all be serialized since they either are related to different service requestors or are serialized anyway by the service requestor. Thus, they can all be processed by a single thread, and a buffer with an unlimited capacity in form of an *active queue* is sufficient. In practice, only a very small number of entries accumulate since the R3 message is sent only in case a service provider gets destroyed. Thus, a component has to buffer R3 messages only for those of its service requestors that are either connected to or are currently trying to get connected to that disappearing service provider.

The *active queue* is placed inside the component management and provides a component centralized decoupling for *server initiated disconnect* messages. The upcall of the R3 message enqueues the request. Each entry in the active queue contains the *void*-casted *this*-pointer of the service requestor instance, the address of the static callback method that has to be invoked to finally call the *disconnect* procedure and the *connection identifier*. The *void*-casted *this*-pointer is needed due to the static callback method. The static callback method allows to use a single active queue to handle the *server initiated disconnects* of all service requestors and independently of their individual template bindings. The upcall of the R3 message is always able to enqueue further R3 messages and gets blocked only for the short time the queue is not accessible due to a currently active read operation.

The thread of the active queue simply blocks on the queue until a new entry is available. It dequeues an entry as soon as there is one available. The first step is to acquire the mutex M100 that protects the *list of registered service requestors (LORSR)* from being modified while a queue

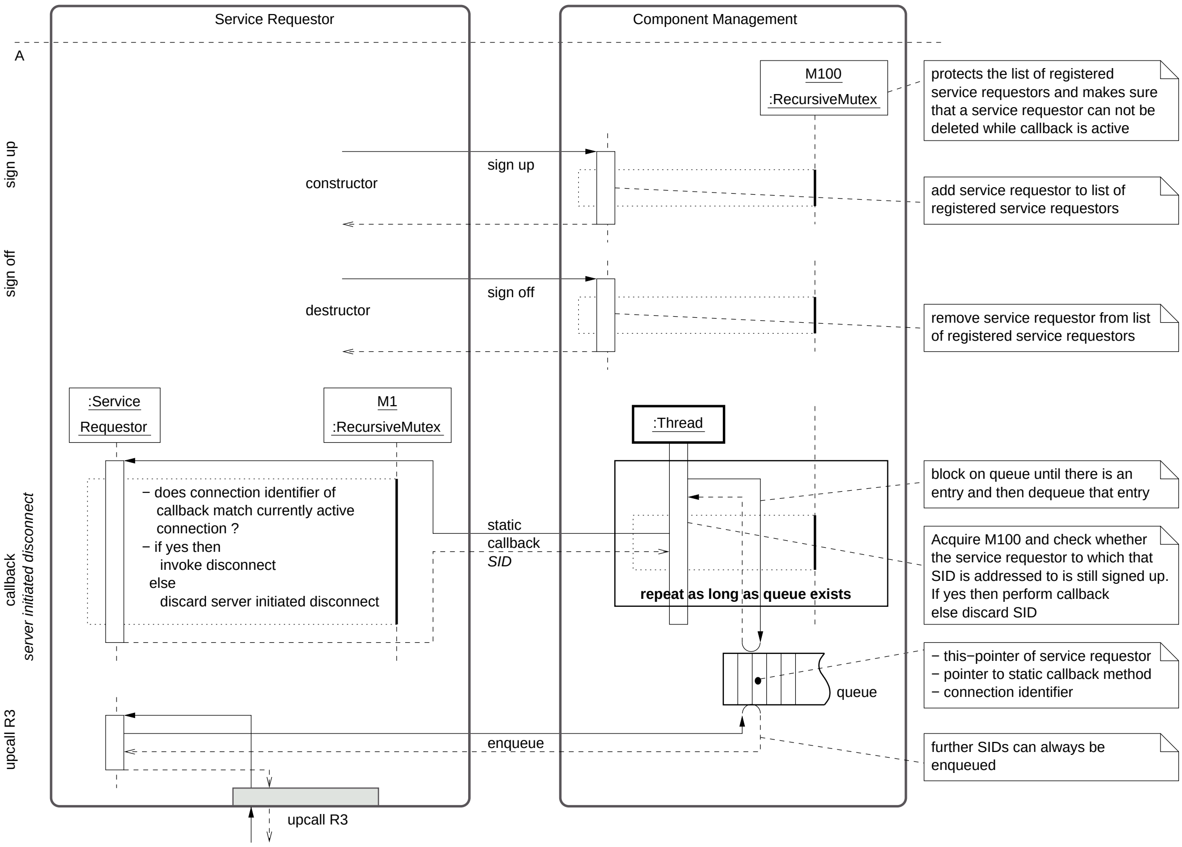

Service Requestor
Component Management
A
sign up
sign off
callback
server initiated disconnect
upcall R3
M100
:RecursiveMutex
protects the list of registered service requestors and makes sure that a service requestor can not be deleted while callback is active
constructor
sign up
add service requestor to list of registered service requestors
destructor
sign off
remove service requestor from list of registered service requestors
:Service Requestor
M1
:RecursiveMutex
:Thread
block on queue until there is an entry and then dequeue that entry
− does connection identifier of callback match currently active connection ?
− if yes then
invoke disconnect
else
discard server initiated disconnect
static
callback
SID
Acquire M100 and check whether the service requestor to which that SID is addressed to is still signed up. If yes then perform callback else discard SID
repeat as long as queue exists
queue
− this−pointer of service requestor
− pointer to static callback method
− connection identifier
enqueue
further SIDs can always be enqueued
upcall R3

Figure 5.104: *The service requestor part of the* server initiated disconnect *procedure.*

entry is processed. The next step is to check whether the entry belongs to a still existing service requestor by means of the *LORSR* since one has to ensure that the callback address is still valid. In case the service requestor still exists, the static callback method is invoked. Since the mutex M100 is not released meanwhile, the service requestor cannot get deleted between checking its existence and invoking and executing the callback method. Within the callback method at the service requestor, the *void*-casted *this*-pointer is casted into the original type and then allows to access the internals of the service requestor instance from inside the static callback method. The callback method first checks whether there is an active connection and invokes a *disconnect* only if the *connection identifier* of the *server initiated disconnect* matches the currently active connection and discards the *server initiated disconnect* otherwise. The *connection identifier* prevents a newly established connection from getting disconnected due to a delayed *server initiated disconnect*. A delayed *server initiated disconnect* can occur in case the service requestor already disconnects itself and independently of any enqueued *server initiated disconnect*. By the way, a *server initiated disconnect* that was sent after the R0 message arrived at the service provider but before the R1 message arrived there, is also correctly discarded due to the not matching *connection identifier*. That, however, presumes that a *connection identifier* is never reused even in case the *connect* failed. Since the mutex M100 is released earliest after the thread of the active queue returns from the callback of the service requestor, no sign off can proceed while a callback is active. Thus, as soon as the sign off was successful, one knows that there is neither an active callback nor can there be any further callback.

Destroying a Service Requestor In case the service requestor is exposed as port at the *wiring slave*, it first resigns from the *wiring slave*. At the service requestor, the response A2 is the latest message after which no more relevant messages are received until a new connection to a service provider is established. The response A2 is the final acknowledgment that the connection to the service provider was successfully closed. However, further messages can still arrive at the service requestor that result from failed attempts to connect to other service providers. In case a timeout occurs when awaiting the response A0 even though the service provider exists, the outdated A0 message still arrives at the service requestor. Even a R3 message can be sent by the service provider after the R0 message arrived and before the R1 message arrived there. The R3 message can be ignored since the address of the service requestor gets removed by the R1 message without requiring a *disconnect* from the service provider. The important property is that those messages are all outdated and thus do not need to be processed. Therefore, it is uncritical to discard them and one can delete buffers without having to care about the remaining entries. In case the service requestor was connected to a service provider, the response A2 of the *disconnect*, that corresponds to the closing of the latest successfully established connection, indicates the final message that has to be processed. All messages following that A2 response can be discarded. In case the service requestor never had been successfully connected to a service provider, there is no need for a *disconnect*. Since there can be only outdated messages from failed attempts to connect to a service provider, there are also no relevant messages that have to be processed before destroying any buffers.

Although one knows when there can be no more relevant messages, one again has to be careful when shutting down the interface object that is labeled by ② in figure 5.80. One has to make sure that the interface object is not deleted while an upcall due to an outdated message is performed from the interface object to the service requestor. For the interface object ②, the same procedure that was already explained with the interface object ④ of the service provider, is applied. Again, all messages not yet processed can be ignored safely. Once the interface object ② is destroyed, no further upcalls into the service requestor can be invoked.

As next, the service requestor signs off from the component management to make sure that no further callbacks due to a still enqueued *server initiated disconnect* are invoked. Since the service requestor is already disconnected, still enqueued messages can simply be ignored. As soon as the service requestor is removed from the *LORSR*, the component management does not invoke the invalidated callback address anymore.

Of course, the interface object ② is destroyed only after the service requestor is disconnected. Thus, the callback detects that its *connection identifier* does not match and the *disconnect* is not invoked from inside the callback. Therefore, it does not matter that the interface object ② is already deleted even though there are still callbacks active until the service requestor signs off. Finally, it is safe to destroy the service requestor.

Concurrency of a *Connect,* a *Disconnect* and a *Server Initiated Disconnect* Both parts of a communication pattern, the service requestor and the service provider, operate independently of each other. Thus, a *server initiated disconnect* can be invoked at the service provider while either a *connect* or a *disconnect* is active or vice versa. The described connection management ensures that any kind of interleaving never results in a transient state.

At the service requestor, all interactions of the connection management are serialized. Most important, a *server initiated disconnect* results in a regular *disconnect* that behaves just like any user invoked *disconnect*. All connects and disconnects always get serialized independently of whether they are invoked from the user level or due to the request of the service provider and due to the *con-*

nection identifier, outdated requests to perform a disconnect can be discarded. Thus, there are no additional aspects over the already described mechanisms of the connection management at service requestors that have to be considered.

At the service provider, the situation is a bit more tricky since the mutex M10 cannot be hold over the whole *server initiated disconnect* procedure. Thus, the service provider has to cope with interleaved upcalls that get processed while the *server initiated disconnect* procedure is worked off. In all stages, one has to make sure that the service provider never leaves a service requestor behind that awaits a response not arriving anymore.

All attempts to connect to a service provider before the point of time labeled by ① in figure 5.103 are regularly accepted and are thus regularly asked to disconnect themselves from the service provider. In ②, the mutex M10 delays further upcalls until all currently connected service requestors are asked to get disconnected ③. In case a R1 message is delayed until the mutex M10 is released, the *list of clients* contains the address of a not connected service requestor. With respect to the service requestor, the *server initiated disconnect* is uncritical since it gets discarded due to the outdated *connection identifier*. With respect to the service provider, the discarded R3 message is uncritical since that entry gets removed as soon as the R1 message is processed after the mutex M10 is released. All attempts to connect after ④ are already rejected but all other interactions are still handled regularly. Thus, new attempts to connect fail so that these service requestors cannot invoke further interactions that would have to be processed. Since each regularly connected service requestor invokes a *disconnect*, the interactions of service requestors get shut down properly one after the other. As soon as the *list of clients* is empty, there are no more pending interactions related to that service provider at any of its formerly connected service requestors nor are there any pending interactions at the service provider. All attempts to connect after ⑤ run into a timeout at the service requestor and thus do not need any support from the service provider to get resolved.

5.6.6.11 The Mapping of the *Connection Oriented Split Protocol*

For the *connection oriented split protocol*, (C*/U) interactions with a *reliable send* policy are sufficient as interface to the underlying communication system. The connection management ensures that connections are established and removed such that all covered interactions can rely on a *delivered* guarantee that is successfully sent messages can never miss their recipient and interactions awaiting a response are always properly resolved and are never left behind. Since the connection management is handled inside the communication patterns, buffers inserted before the communication patterns that convert a *processed* policy into a *reliable send* policy are not a problem anymore. Since the administrative interactions do not depend on a decoupling, the decoupling has to be achieved for the service related interactions only. Service related requests or responses that depend on a decoupling are all forwarded to user level handlers. The decoupling mechanisms and the corresponding threading models can thus be provided inside the appropriate handler types. Service related requests or responses that do not invoke a user level handler and that are processed inside the communication patterns always only overwrite already hold buffers and broadcast a signal to release pending method calls but never invoke the actual processing. Thus, all service related upcalls that remain in the communication patterns do also not depend on a decoupling mechanism. The only exception is the service requestor part of the *server initiated disconnect*. However, that decoupling is implemented efficiently by means of a component central mechanism. Thus, the *connection oriented split protocol* can be mapped easily and efficiently onto all types of communication mechanisms.

Figure 5.105 summarizes the mapping of the (C*/U) interaction patterns of the communication patterns. In principle, all service providers have to be independent of each other so that they cannot

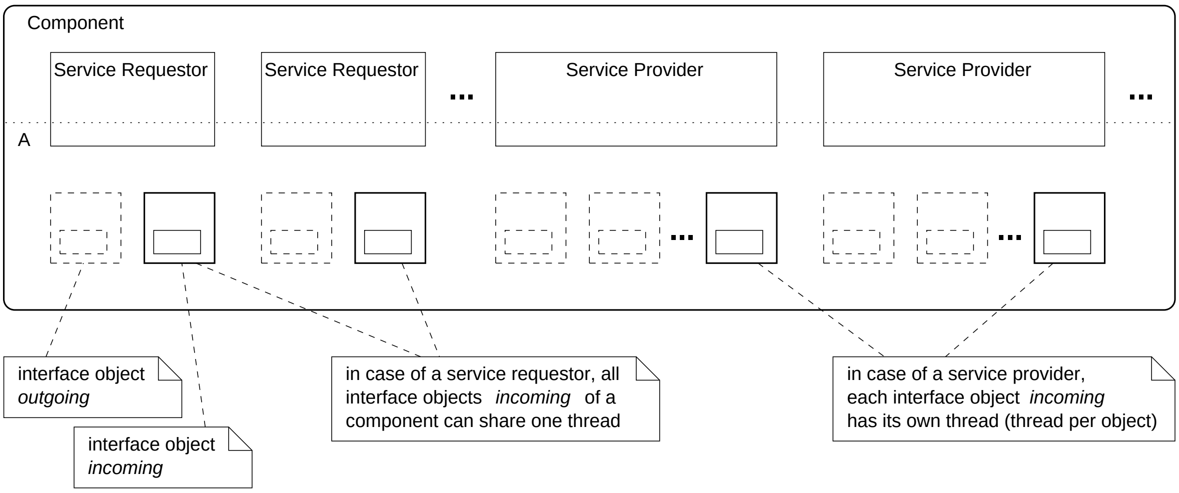

Component
Service Requestor
Service Requestor
...
Service Provider
Service Provider
...
A
...
...
interface object *outgoing*
interface object *incoming*
in case of a service requestor, all interface objects *incoming* of a component can share one thread
in case of a service provider, each interface object *incoming* has its own thread (thread per object)

Figure 5.105: *The mapping of the (C*/U) interaction patterns of the* connection oriented split protocol.

share any resources, neither threads or buffers nor sockets or mailboxes or servant objects. The reason is that otherwise all service providers sharing resources would get blocked altogether in case of a tailback caused by one of the service providers. Thus, each service provider always has its own socket and mailbox and servant object, respectively. Furthermore, at a service provider, the interface object for incoming messages is always operated by a private thread. A single thread is sufficient since serialization is allowed. In case of an object based middleware, a *thread per object* model is selected.

In contrast thereto, the upcalls from the communication system into service requestors are always able to forward all their messages. Service requestors look like sinks with an unbounded capacity. Thus, the interface objects for incoming messages of all service requestors of a component can share the same resources. Of course, it finally depends on the available resources whether it is worthwhile to implement the interface objects such that they share a mailbox or a socket, for example. In principle, a single thread and a single communication port like a mailbox, a socket or a servant object is sufficient to handle the incoming messages of all service requestors of a component. Normally, though, the interface objects of the service requestors show the same structure as those of the service providers. Thus, each interface object for incoming messages, independently of whether it belongs to a service requestor or to a service provider, possesses its communication port and its thread to operate the interface object.

The (C*/U) interactions of the *connection oriented split protocol* can be directly mapped onto the interaction model (F)* without requiring any decoupling beneath the level of the communication patterns. Besides the *server initiated disconnect*, only such interactions depend on a decoupling mechanism or on threading models that also invoke user level handlers. Thus, the required decoupling and threading mechanisms can be implemented inside the handlers so that they are still not visible to the user but are located even above the communication patterns. Simply by adding further handler types, one can extend the range of available threading models without having to modify the communication patterns. However, it is important to activate a *thread per object* policy with the servant objects as described above. On top of the interaction model (F)* with its *processed* policy and its *synchronous* server part invocation mode, more advanced threading models of the communication system need not to be activated since these nevertheless require the user level decoupling. Otherwise, the *delivered* characteristic of the client parts of the interaction patterns is violated. Due to the connection management, one could even insert buffers above the communication system without running into troubles due to misinterpreted acknowledgments. With respect to the mapping, there is no difference between the standard synchronous (F)* interactions of *CORBA* and those of *RPC* mechanisms.

One approach to fully explore the threading models of *CORBA* with its synchronous server part interface is to use the *AMI* model. Then, however, one not only depends on the current stage of the *AMI* implementations with their bulky interfaces but also on the available threading models. Furthermore, threading models are always enabled on a per-servant basis so that even administrative interactions get their own thread in case of a *thread per invocation* policy, for example.

Due to the *connection management*, there is now nothing special about the (G2) interaction model. Thus, message based systems do not anymore impose severe mapping problems and the mapping of the ($C^{\star}$/U) interaction pattern is straightforward. With *CORBA*, one does not anymore depend on the *sync with server* policy of *oneway* declared methods and one can even use a weaker policy that corresponds to a *reliable send* only. In particular with *mailboxes*, one does not anymore have the problem of being allowed to shutdown a mailbox only after all its entries have been removed. Of course, the same holds true for the buffer of the interaction model (E). Due to the connection management, all kinds of buffers that result in a *reliable send* policy do impose any problems anymore, neither in *mailboxes* nor on top of *TCP sockets*.

5.6.7 The Communication Patterns

The following sections give detailed insights into the internal structures of every single communication pattern. Of course, neither the administrative interactions nor the underlying interaction patterns need to be explained anymore. So far, however, only the basic principles of the interplay of the service related and the administrative interactions have been illustrated. Thus, the focus is now put onto the service related interactions and their individual interplay with the administrative interactions.

5.6.7.1 The Component Management Class

The basic interactions of the communication patterns with the component infrastructure are summarized in figure 5.106. The component management operates the communication mechanism of the component and provides the access to the communication infrastructure. In case of the *CORBA* based implementation, the overall infrastructure consists of the *ORB*, the *POA* and the access to the *name service*. Furthermore, it provides the name of the component to the service providers of the component.

The component management hosts the decoupling mechanism needed by the *server initiated disconnect* procedure. The interface at the component management class consists of the *sign up*, the *sign off* and the *enqueue* methods and the service requestor provides the *callback sid* method that is invoked by the component management class. The first argument of the *callback sid* method of the service requestor is the *void*-casted *this*-pointer and the second argument is the *connection identifier*. The *callback sid* method of the service requestor and the *enqueue* method of the component management class are already illustrated in figure 5.104.

Figure 5.107 shows the details of the component management class. The timer service is for example used by the *push timed* service provider to get notified whenever a new update is due. A component central timer is more efficient than having one in every pattern instance. The mutex M100 not only coordinates the access to the *list of registered service requestors (LORSR)* but also makes sure that a service requestor cannot disappear while the component management invokes a callback

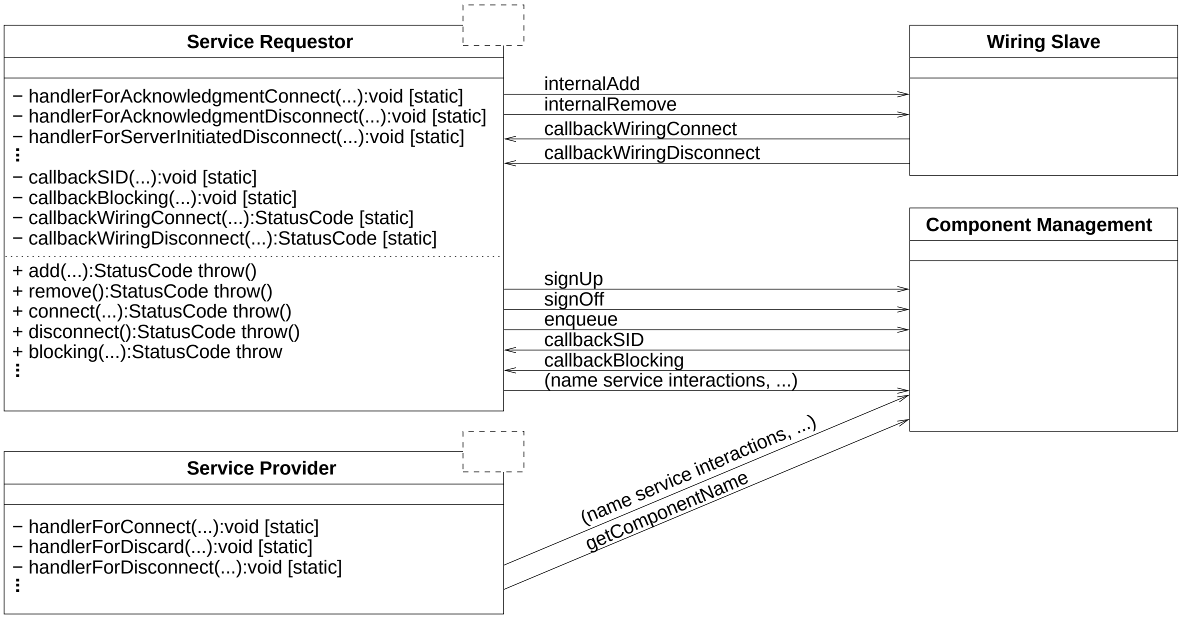

Service Requestor
− handlerForAcknowledgmentConnect(...):void [static]
− handlerForAcknowledgmentDisconnect(...):void [static]
− handlerForServerInitiatedDisconnect(...):void [static]
⋮
− callbackSID(...):void [static]
− callbackBlocking(...):void [static]
− callbackWiringConnect(...):StatusCode [static]
− callbackWiringDisconnect(...):StatusCode [static]
+ add(...):StatusCode throw()
+ remove():StatusCode throw()
+ connect(...):StatusCode throw()
+ disconnect():StatusCode throw()
+ blocking(...):StatusCode throw
⋮
Wiring Slave
internalAdd
internalRemove
callbackWiringConnect
callbackWiringDisconnect
Component Management
signUp
signOff
enqueue
callbackSID
callbackBlocking
(name service interactions, ...)
Service Provider
− handlerForConnect(...):void [static]
− handlerForDiscard(...):void [static]
− handlerForDisconnect(...):void [static]
⋮
(name service interactions, ...)
getComponentName

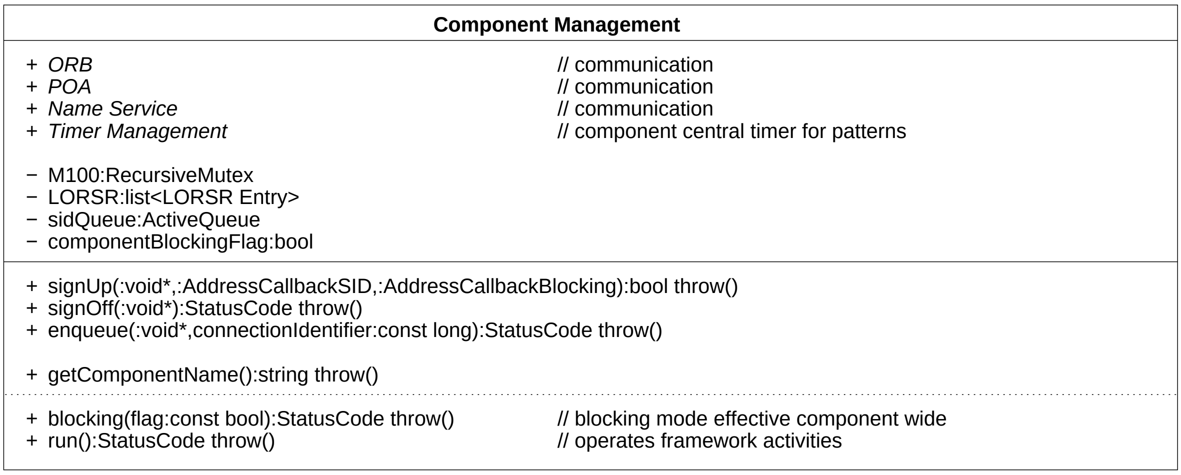
Figure 5.106: *Basic interactions of communication patterns with the component infrastructure.*

at the service requestor. As long as the mutex M100 is hold, no service requestor can sign off and a service requestor gets destroyed only after it signed off.

Besides their role with respect to the *server inititated disconnect*, the mutex M100 and the *LORSR* are also used to implement the component wide effective blocking mode. The blocking method of the component management class needs to be able to invoke the *blocking component* method of all service related monitors. Since service providers currently solely offer non-blocking member functions at the user interface, only service requestors possess service related monitors that need to be accessed by the component management. Thus, service requestors provide the *callbackBlocking* method that can be invoked from the component management to iterate through all service related monitors of a service requestor. The first argument again is the *void*-casted *this*-pointer and the second argument contains the new state of the *component blocking flag* that is forwarded to every service requestor. The *LORSR* already provides the access to all service requestors of a component and the mutex M100 ensures that a service requestor does not disappear while its *callback blocking* method is invoked.

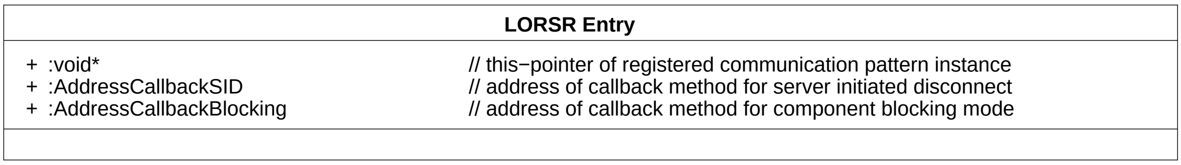
Figure 5.108 shows the entry of the *LORSR*. The *this*-pointer identifies the communication pattern instance and is also returned with every callback to a communication pattern instance. The next two entries contain the address of the callback for the *server initiated disconnect* and that of the callback of the blocking mode. A new entry is generated by the *sign up* method and it is removed by the *sign off* method. By the way, service providers can use the same *sign up* and *sign off* methods in case they once possess service related monitors. There are no modifications needed inside the component management class. However, service providers also need to be extended by an appropriate *callback blocking* method. The address of the *callback sid* method is set to *null*. Since a service provider never enqueues a R3 message, that address is never invoked by the active queue of the component management class.

Figure 5.109 shows the details of the *sign up* and the *sign off* procedure. The *sign up* method provides all arguments required to generate a new entry in the *LORSR* and it returns the current state



```
Component Management

+ ORB                                         // communication
+ POA                                         // communication
+ Name Service                                // communication
+ Timer Management                            // component central timer for patterns

− M100:RecursiveMutex
− LORSR:list<LORSR Entry>
− sidQueue:ActiveQueue
− componentBlockingFlag:bool

+ signUp(:void*,:AddressCallbackSID,:AddressCallbackBlocking):bool throw()
+ signOff(:void*):StatusCode throw()
+ enqueue(:void*,connectionIdentifier:const long):StatusCode throw()

+ getComponentName():string throw()

+ blocking(flag:const bool):StatusCode throw()   // blocking mode effective component wide
+ run():StatusCode throw()                       // operates framework activities
```

Figure 5.107: *The framework builder view on the component management class.*



```
LORSR Entry

+ :void*                          // this−pointer of registered communication pattern instance
+ :AddressCallbackSID             // address of callback method for server initiated disconnect
+ :AddressCallbackBlocking        // address of callback method for component blocking mode
```

Figure 5.108: *The class diagram of an entry of the* LORSR.

of the *component blocking flag* of the component management that gets stored in the communication pattern. The *component blocking flag* of the communication patterns is used to properly initialize dynamically generated monitors that would otherwise not know about the current state of the component blocking mode. At a service requestor, that flag is protected by the mutex M2. The mutex M2 is hold during the *sign up* procedure since otherwise it might happen that the *callback blocking* method gets invoked at ① so that the *component blocking flag* gets overwritten afterwards by the meanwhile outdated return value. It is important to note that holding the mutex M2 when invoking the *sign up* method does not cause a deadlock. In case the *sign up* method gets blocked on the mutex M100 at ②, it is not yet in the list since the *sign up* is called only once per service requestor so that it does not get a callback that needs the mutex M2. In case the blocking mode of the component management gets the mutex M100 at ③, the callback gets blocked on the mutex M2. That, however, is at ① after the *sign up* already returned so that the release of the mutex M2 does not anymore depend on the *sign up* method. In contrast to the *sign up* method, the mutex M2 must not be hold while the *sign off* method is invoked since now, the service requestor is already contained in the *LORSR* so that callbacks can be invoked on the service requestor. Since the *sign off* does not return the initial state for the *component blocking flag* of the service requestor, there is also no need to hold the mutex M2.

Figure 5.110 shows the details of the *blocking* method of the component management. It iterates through the *LORSR* and invokes the *callback blocking* method for all registered communication pattern instances. The callback iterates through the *list of monitors* and invokes the *blocking component* method on every monitor.

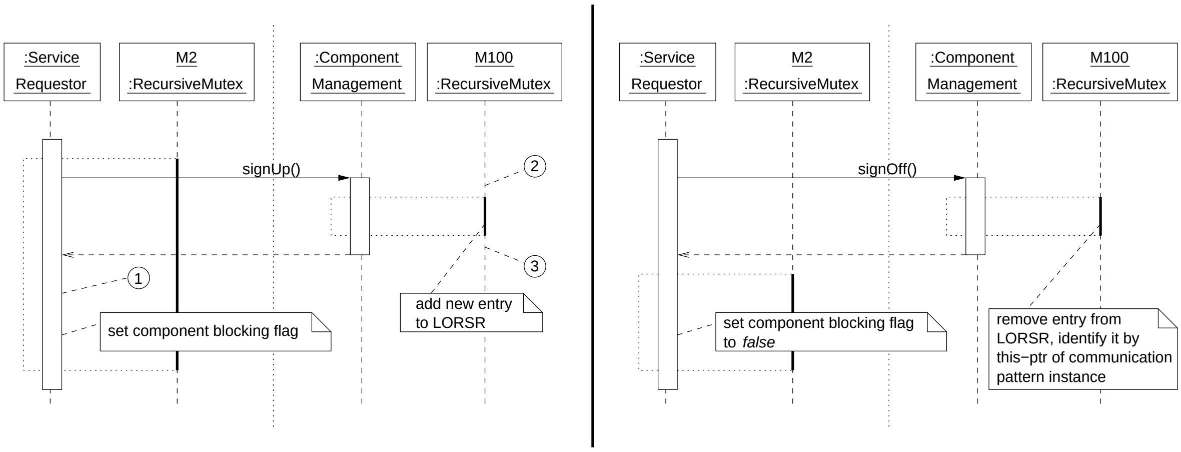

:Service Requestor
M2 :RecursiveMutex
:Component Management
M100 :RecursiveMutex
signUp()
2
3
1
add new entry to LORSR
set component blocking flag
signOff()
set component blocking flag to *false*
remove entry from LORSR, identify it by this−ptr of communication pattern instance

***Figure 5.109:** The sequence diagrams of the* sign up *and* sign off *procedures.*

Hint on the Callback Interface The described callback interface that is based on *static* callback methods is an implementational detail of the *C++* based implementation. A static callback method always needs the *void*-casted *this*-pointer as its first argument. Inside the callback method, the original type of the communication pattern instance is known so that the *void* argument can be casted into the original type. The *this*-pointer provides the access to the communication pattern instance from inside the static method. The advantage of this approach is that one can easily manage callback methods of different classes even if these are not derived from a common base class. Earlier implementations suffered from flawed template implementations so that one could not use a common base class for all communication pattern templates. Even though these flaws do not exist anymore with up-to-date compilers, the callback interface has not been changed anymore. In particular with the interface objects, that kind of callback mechanism approved to be very useful and still simplifies the overall implementation. Since these details are only visible to the framework builder, one can ensure that the callback methods are only used in the intended way. On the basis of the described details, it is not only possible to implement the communication patterns by means of other programming languages, but one could also easily refactor the callback interface.

5.6.7.2 The Send Pattern

The *send* pattern is the most simple communication pattern. It does even not need all the structures introduced in the previous sections. In particular, the mutex M2 is not required since there are neither deferred answers nor does the *send* pattern possess any service related interactions that need to be informed about any state changes. Thus, the basic structures that can be found in any of the communication patterns, are explained by means of the *send* pattern.

The Client Part The client side internals of the *send client* class are shown in figure 5.111. The methods comprise the already known user interface methods as well as the callback methods for incoming messages. The arguments of the latter conform to the message format with the first argument being the already known *void*-casted *this*-pointer of the *send client* instance. The set of attributes of the *send client* pattern comprises the minimum number of required attributes. These attributes can therefore also be found in the other client patterns.

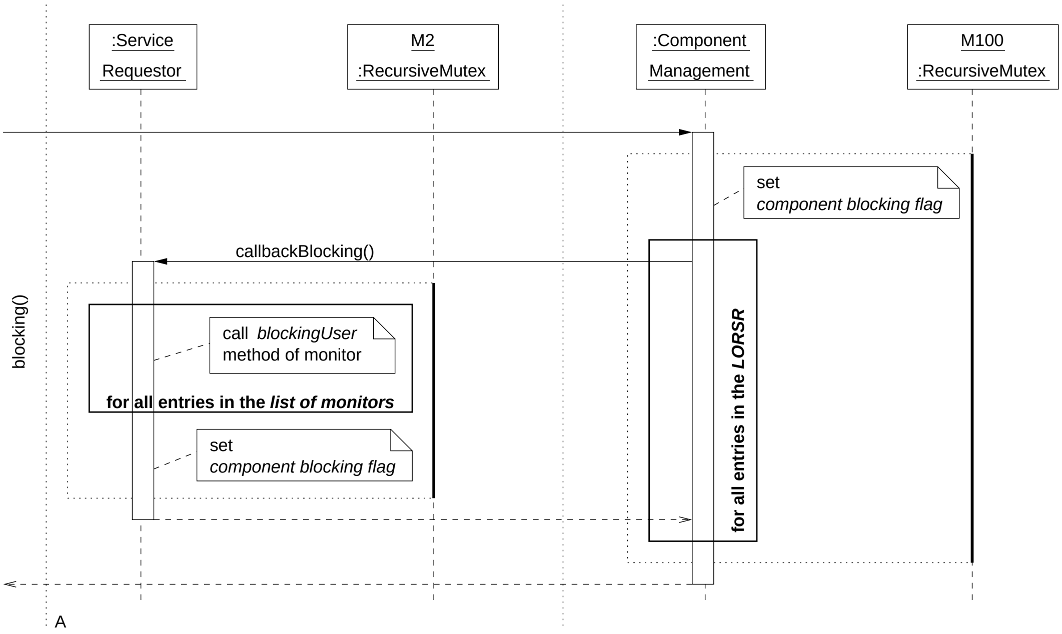

:Service Requestor
M2 :RecursiveMutex
:Component Management
M100 :RecursiveMutex
set component blocking flag
callbackBlocking()
blocking()
call blockingUser method of monitor
for all entries in the list of monitors
set component blocking flag
for all entries in the LORSR
A

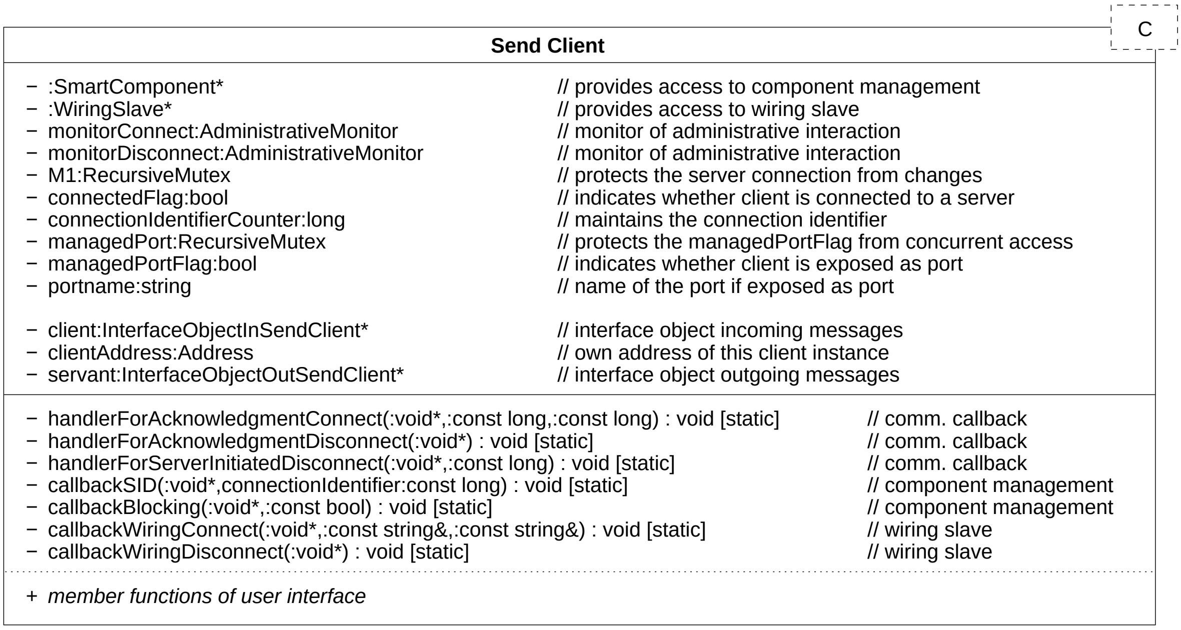
Figure 5.110: *The sequence diagrams of the component* blocking *method.*

Table 5.111 gives further explanations on some of the attributes. The last three attributes of the *send client* provide the interface to the communication system. The *client* is the interface object for incoming messages that forwards incoming messages to the handlers of the *send client*. The *clientAddress* is the address of the *send client* that is transmitted to service providers in case of a connect. The *servant* is the interface object for outgoing messages and is available only in case the *send client* is connected to a service provider.

The Client Side Administration Internals The client side administration interface comprises the *add, remove, connect, disconnect* and the *blocking* method. The client side parts of the *connect*, the *disconnect* and the *server initiated disconnect* of the *send* client comply with the generic form of those administrative interactions. The only difference is that all parts related to the mutex M2 and the

Attribute	Description
connectedFlag:bool	Indicates whether a connection to a service provider exists. If *true*, one can send messages to the now valid server address. This flag is protected by the mutex M1.
connectionIdentifierCounter:long	Is incremented with every attempt to *connect* to a service provider and allows to identify outdated requests to get disconnected and outdated responses due to failed attempts to connect to a service provider. It is also protected by the mutex M1.
managedPort:RecursiveMutex	This mutex is used to protect the *managed port flag* and to synchronize the execution order of concurrent calls to the *add / remove* methods.
managedPortFlag:bool	Indicates whether that service requestor is registered as port at the wiring slave. In that case, the *portname* holds a valid port name.

Table 5.47: *Description of the client part attributes.*



```
C
Send Client
− :SmartComponent*                                  // provides access to component management
− :WiringSlave*                                     // provides access to wiring slave
− monitorConnect:AdministrativeMonitor              // monitor of administrative interaction
− monitorDisconnect:AdministrativeMonitor           // monitor of administrative interaction
− M1:RecursiveMutex                                 // protects the server connection from changes
− connectedFlag:bool                                // indicates whether client is connected to a server
− connectionIdentifierCounter:long                  // maintains the connection identifier
− managedPort:RecursiveMutex                        // protects the managedPortFlag from concurrent access
− managedPortFlag:bool                              // indicates whether client is exposed as port
− portname:string                                   // name of the port if exposed as port

− client:InterfaceObjectInSendClient*               // interface object incoming messages
− clientAddress:Address                             // own address of this client instance
− servant:InterfaceObjectOutSendClient*             // interface object outgoing messages

− handlerForAcknowledgmentConnect(:void*,:const long,:const long) : void [static]     // comm. callback
− handlerForAcknowledgmentDisconnect(:void*) : void [static]                          // comm. callback
− handlerForServerInitiatedDisconnect(:void*,:const long) : void [static]             // comm. callback
− callbackSID(:void*,connectionIdentifier:const long) : void [static]                 // component management
− callbackBlocking(:void*,:const bool) : void [static]                                // component management
− callbackWiringConnect(:void*,:const string&,:const string&) : void [static]         // wiring slave
− callbackWiringDisconnect(:void*) : void [static]                                    // wiring slave

+ member functions of user interface
```

Figure 5.111: *Details of the internals of the* send client *class.*

list of monitors are removed. Since the *send* pattern has no blocking methods, the *blocking* method is available for compatibility reasons only and simply does nothing. Therefore, the *callback blocking* method is also just empty.

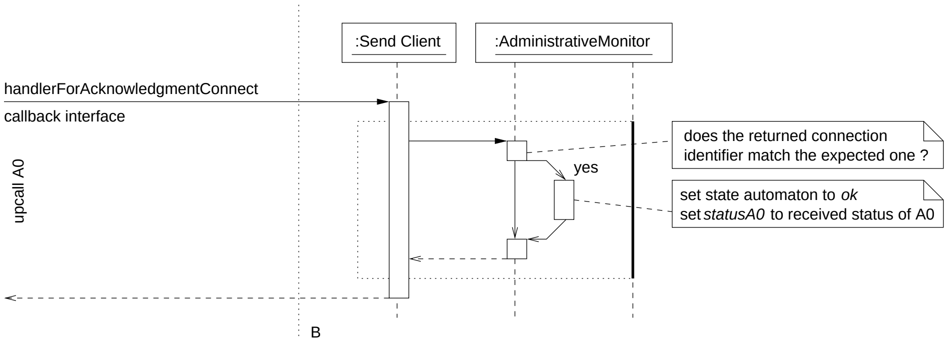

:Send Client
:AdministrativeMonitor
handlerForAcknowledgmentConnect
callback interface
upcall A0
yes
does the returned connection identifier match the expected one ?
set state automaton to *ok*
set *statusA0* to received status of A0
B

Figure 5.112: *The internals of the client side handler for the* acknowledgment message A0.

The internals of the handler for the acknowledgment message A0 are shown in figure 5.112. The upcall acquires the mutex of the administrative monitor and checks whether the returned connection identifier matches the expected one. If that holds true, the state of the automaton is set to *ok*, the received status is stored in the *statusA0* attribute so that it becomes accessible from inside the *send client* object and finally, a broadcast is emitted before the mutex of the monitor is released. Due to the connection identifier, outdated A0 messages are properly discarded.

As already shown in figure 5.106, service requestors can add themselves to a *wiring slave* to become an externally visible port. Thus, each service requestor possesses an *add* and a *remove* method and also provides the *callback wiring connect* and the *callback wiring disconnect* methods to execute wiring requests from the wiring slave. The callback methods simply invoke the regular *connect* and *disconnect* methods. The wiring slave can directly invoke the regular *connect* and *disconnect* method in case all service requestors are derived from a common base class that already provides the two methods. Then, the callback methods are not needed anymore.

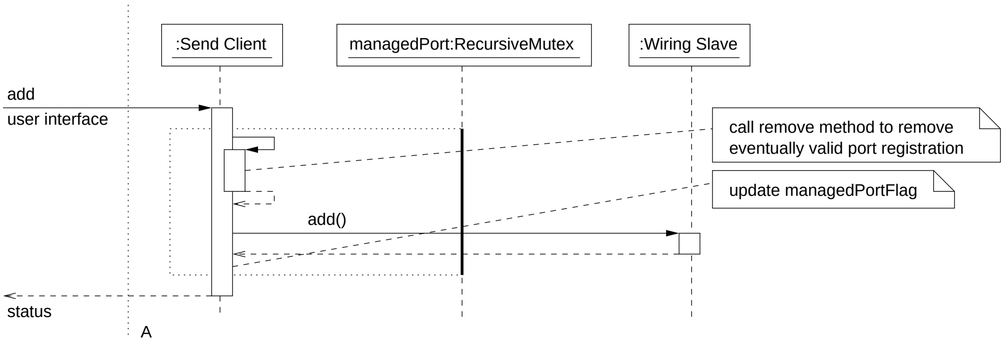

:Send Client
managedPort:RecursiveMutex
:Wiring Slave
add
user interface
call remove method to remove eventually valid port registration
update managedPortFlag
add()
status
A

Figure 5.113: *The internals of the client side* add *method.*

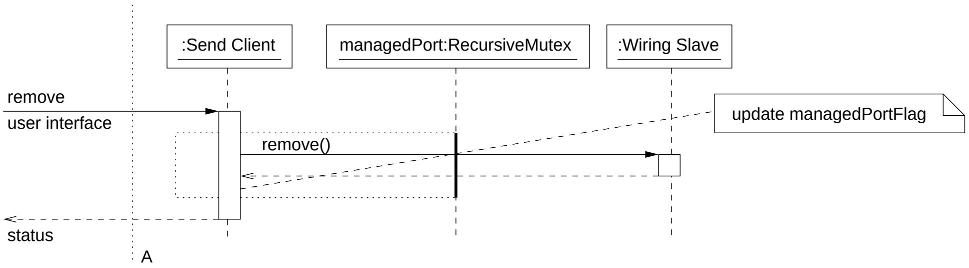


Figure 5.114: *The internals of the client side* remove *method.*

The *add* method is shown in figure 5.113 and the *remove* method in figure 5.114. The *managedPort* mutex is used to serialize concurrent calls to the *add* and *remove* methods since otherwise the port state could be messed up in the service requestor.

The wiring slave maintains a *list of ports* that is protected by a mutex. That mutex is hold by the wiring slave while it calls the *connect* or the *disconnect* method of a service requestor. Due to the mutex, a service requestor cannot get removed from the *list of ports* while the wiring slave is accessing it. Thus, it can also not get destroyed while the wiring slave is acting on the service requestor since the service requestor always invokes the *remove* method first. Since both the *connect* and the *disconnect* procedure, do not need to acquire the *managedPort* mutex, an *add* and *remove* method that get blocked on the mutex of the wiring slave and that holds the *managed port* mutex imposes no problems.

The Client Side Service Internals Figure 5.115 shows the *send* method that transmits the user provided communication object to the service provider. A *send* can be performed only if the client is connected to a server. Of course, the *send* operation is protected by the mutex M1.

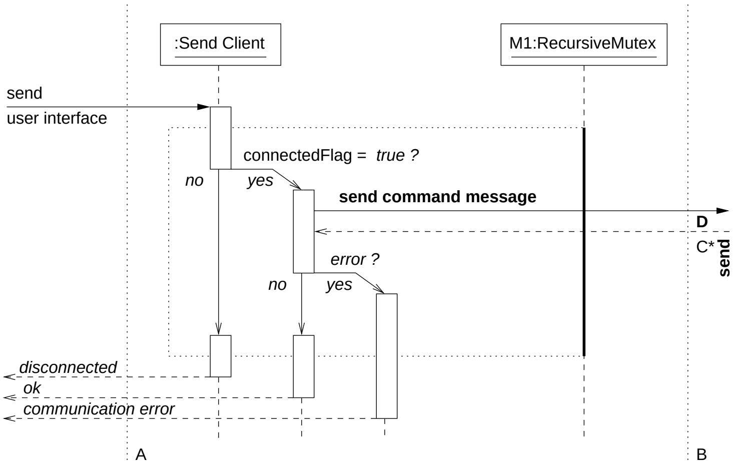
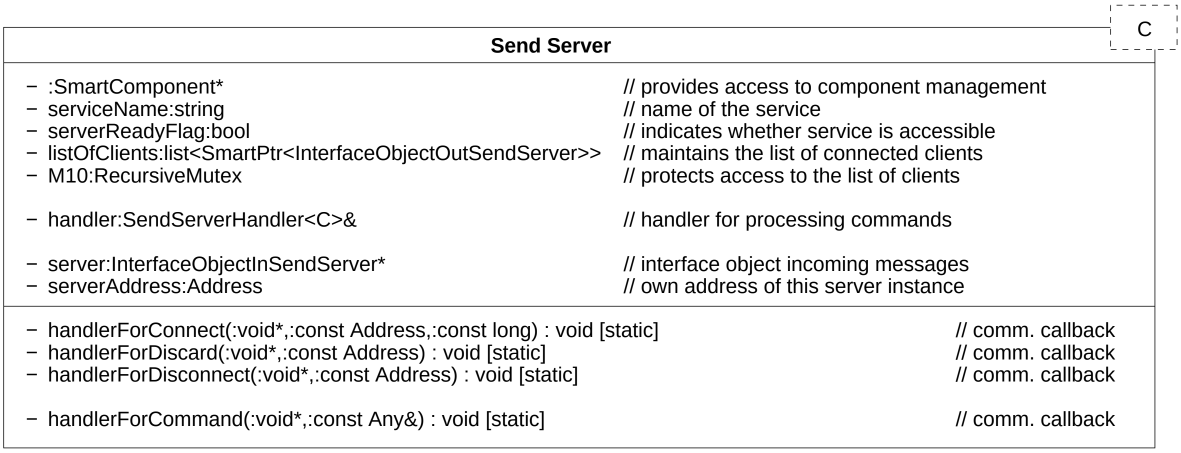


***Figure 5.115:** The internals of the client side* send *method.*

The Server Part The server side internals of the *send server* class are shown in figure 5.116. The set of methods again comprises the callback methods for incoming messages. There are no user interface methods listed since the *send server* possesses only a handler based interface. The set of attributes of the *send server* pattern comprises the minimum number of attributes required in a service provider.



C

Send Server

```
− :SmartComponent*                                         // provides access to component management
− serviceName:string                                       // name of the service
− serverReadyFlag:bool                                     // indicates whether service is accessible
− listOfClients:list<SmartPtr<InterfaceObjectOutSendServer>>  // maintains the list of connected clients
− M10:RecursiveMutex                                       // protects access to the list of clients

− handler:SendServerHandler<C>&                            // handler for processing commands

− server:InterfaceObjectInSendServer*                       // interface object incoming messages
− serverAddress:Address                                    // own address of this server instance

− handlerForConnect(:void*,:const Address,:const long) : void [static]      // comm. callback
− handlerForDiscard(:void*,:const Address) : void [static]                  // comm. callback
− handlerForDisconnect(:void*,:const Address) : void [static]               // comm. callback

− handlerForCommand(:void*,:const Any&) : void [static]                     // comm. callback
```

***Figure 5.116:** Details of the internals of the* send server *class.*

The *server ready* flag indicates whether a service provider is accessible. It is protected by the mutex M10. The *handler* attribute provides access to the handler that processes incoming commands. The *server* is the interface object for incoming messages that forwards incoming messages to the handlers of the *send server*. The *server address* is the address of the *send server* that is registered at the name service. The *list of clients* contains the interface objects for outgoing messages to connected service requestors. With the *send* pattern, that list is only needed to be able to invoke a *server initiated disconnect*.

The Server Side Administration Internals With the *send* pattern, there are no service related interactions that need to be resolved by any of the administrative interactions. Thus, the server parts of the *connect*, the *disconnect* and the *server initiated disconnect* of the *send* server comply with the generic form of those administrative interactions.

The Server Side Service Internals Figure 5.117 shows the server side part of the service related (C/U) interaction. It does not require any server side administrative structures and simply invokes the provided handler. It does not require the mutex M10 since a disconnect of any client can get processed only via the same thread that currently forwards the command.

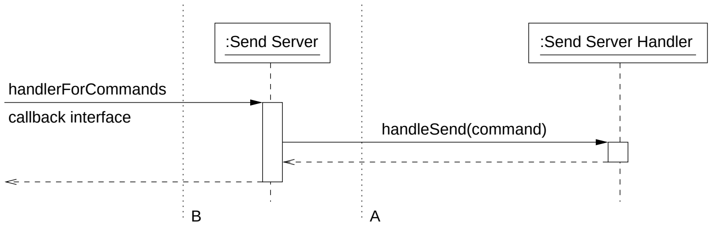
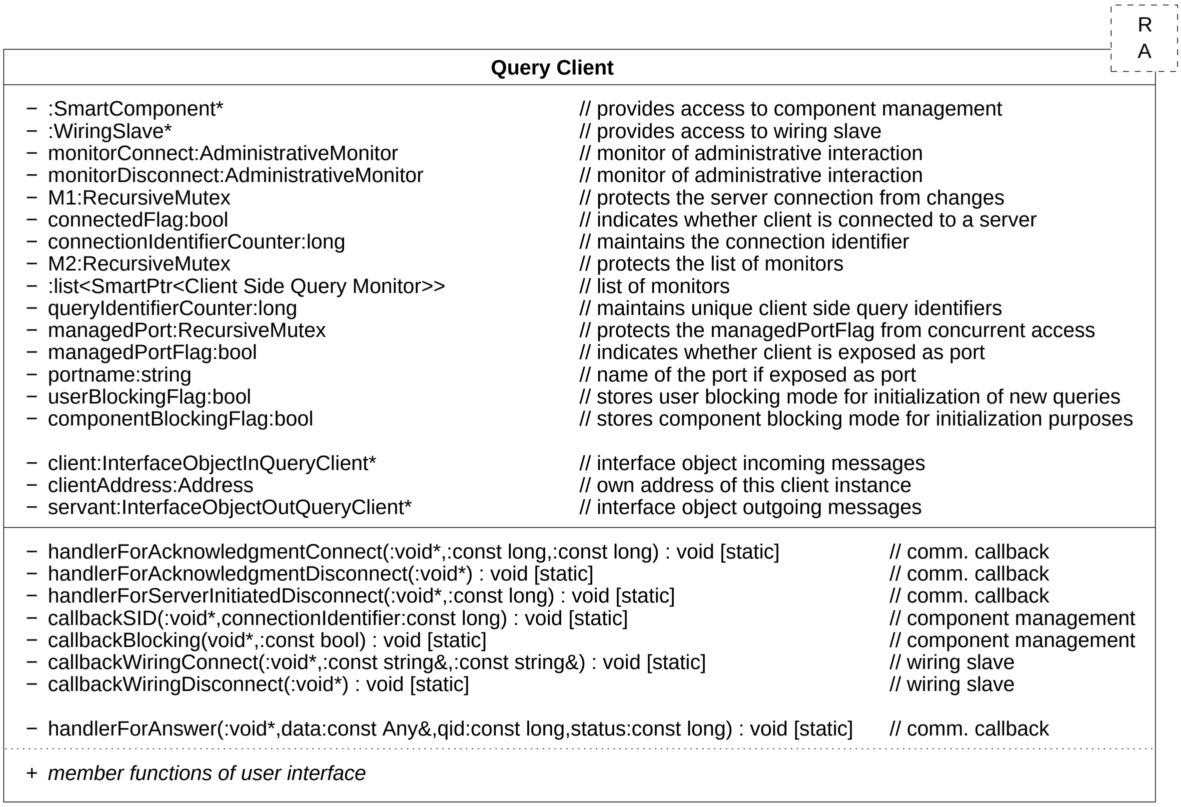


Figure 5.117: *The internals of the server side* command *handler.*

5.6.7.3 The Query Pattern

The *query* pattern is the most complex pattern with respect to the internals since it provides a two-way interaction that is based on an asynchronous request/response protocol. That requires the client side to properly assign the received responses to pending requests and the server side to distribute available answers to the proper clients. Furthermore, disconnecting a client now requires to clean up not yet fully processed requests at the server.

The Client Part The client side internals of the *query client* class are shown in figure 5.118. The additional attributes compared to the *send* pattern are explained in table 5.48. The handler for the answer message accepts the incoming responses. The first argument is the already known *this*-pointer, the second argument the content of the communication object of the answer and the third argument the client side query identifier that was provided with the request. It is returned to be able to correctly assign answers to open requests. The fourth argument indicates whether the response contains a valid response. That is the case if the request was rejected at the service provider or got discarded there.



```
Query Client

− :SmartComponent*                              // provides access to component management
− :WiringSlave*                                 // provides access to wiring slave
− monitorConnect:AdministrativeMonitor          // monitor of administrative interaction
− monitorDisconnect:AdministrativeMonitor       // monitor of administrative interaction
− M1:RecursiveMutex                             // protects the server connection from changes
− connectedFlag:bool                            // indicates whether client is connected to a server
− connectionIdentifierCounter:long              // maintains the connection identifier
− M2:RecursiveMutex                             // protects the list of monitors
− :list<SmartPtr<Client Side Query Monitor>>    // list of monitors
− queryIdentifierCounter:long                   // maintains unique client side query identifiers
− managedPort:RecursiveMutex                    // protects the managedPortFlag from concurrent access
− managedPortFlag:bool                          // indicates whether client is exposed as port
− portname:string                               // name of the port if exposed as port
− userBlockingFlag:bool                         // stores user blocking mode for initialization of new queries
− componentBlockingFlag:bool                    // stores component blocking mode for initialization purposes

− client:InterfaceObjectInQueryClient*          // interface object incoming messages
− clientAddress:Address                         // own address of this client instance
− servant:InterfaceObjectOutQueryClient*        // interface object outgoing messages

− handlerForAcknowledgmentConnect(:void*,:const long,:const long) : void [static]          // comm. callback
− handlerForAcknowledgmentDisconnect(:void*) : void [static]                               // comm. callback
− handlerForServerInitiatedDisconnect(:void*,:const long) : void [static]                  // comm. callback
− callbackSID(:void*,connectionIdentifier:const long) : void [static]                      // component management
− callbackBlocking(void*,:const bool) : void [static]                                      // component management
− callbackWiringConnect(:void*,:const string&,:const string&) : void [static]              // wiring slave
− callbackWiringDisconnect(:void*) : void [static]                                         // wiring slave

− handlerForAnswer(:void*,data:const Any&,qid:const long,status:const long) : void [static]  // comm. callback

+ member functions of user interface
```

Figure 5.118: *Details of the internals of the* query client *class.*

All active queries are represented by their own monitor instance that is shown in figure 5.119. Each query can be identified by the unique query identifier and provides storage for the received answer. The *wait* method is adjusted to the needs of the *query* pattern. It blocks only if the state of the automaton is *pending* since that is the only state in which an answer can still be expected. Of course, the blocking indicator must also evaluate to *true* indicating that blocking is allowed.

Attribute	Description
:list<SmartPtr<Client Side Query Monitor>>	The *list of monitors* holds the monitors of the currently active queries.
queryIdentifierCounter:long	That counter provides the unique identifier assigned to every query. It is protected by the mutex M2.
userBlockingFlag:bool	The current setting of the user blocking mode needs to be stored to be able to properly initialize newly created monitors. It is protected by the mutex M2.
componentBlockingFlag:bool	The current setting of the component blocking mode that gets initialized by the *sign up* method. It is updated by the *callback blocking* method and is protected by the mutex M2.

Table 5.48: *Description of the client part attributes.*

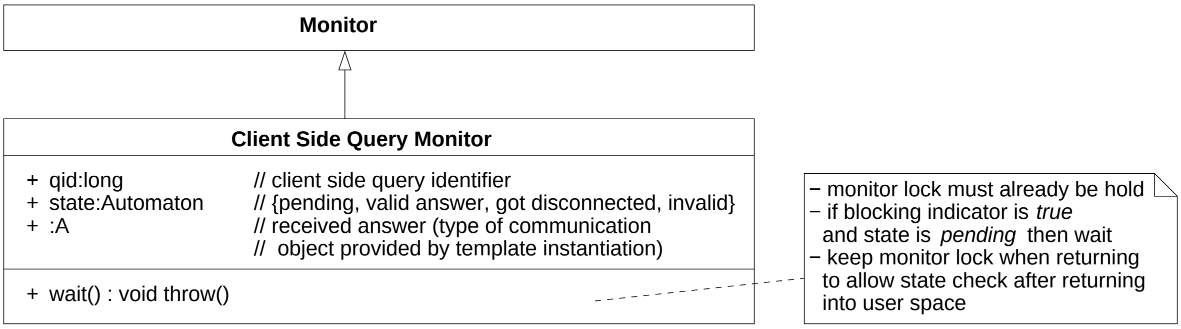

Monitor
Client Side Query Monitor
+ qid:long // client side query identifier
+ state:Automaton // {pending, valid answer, got disconnected, invalid}
+ :A // received answer (type of communication
// object provided by template instantiation)
+ wait() : void throw()
− monitor lock must already be hold
− if blocking indicator is true
and state is pending then wait
− keep monitor lock when returning
to allow state check after returning
into user space

Figure 5.119: *The client side representation of an active query.*

Figure 5.120 shows the client side state automaton of a query. A new request initializes its state automaton to the *pending* state. Receiving the answer performs the state transition *c* to the *valid answer* state. Consuming the received answer by either a *receive*, a *receive wait* or a *discard* performs the state transition *e*. In the final *invalid* state, the monitor is removed from the *list of monitors* which invalidates the query identifier. Due to the used smart pointers, it gets destroyed as soon as no more references point to it. Performing a *disconnect* while the answer has not yet been received performs the state transition *g*. The *got disconnected* state indicates that the answer belonging to this query identifier cannot be received anymore. The state transition *i* is performed when the query identifier gets consumed. Performing a *disconnect* in the *valid answer* state is handled by the state transition *d*. A *disconnect* does not affect already received but not yet fetched answers. These keep valid and can be fetched at any point of time. Finally, a *disconnect* does not affect the *disconnected* state. That is captured by the state transition *h*. The state transition *b* is executed by the *receive* method in case the answer is not yet available and a *discard* performs the state transition *k*.

The important point now is that a broadcast never releases a blocking method wrongly such that it would have to reinvoke the *wait*. The state transitions *c*, *g* and *k* invoke a *broadcast* since leaving the *pending* state either means that the expected answer is available or that it cannot be received anymore or that the query got discarded. In all cases, it makes no sense to further block so that the *wait* does not need to be reinvoked once the *pending* state is left. In case the *wait* awakes in the *pending* state, the *blocking indicator* evaluates to *false* and blocking calls have to return to the user level. Again, the *wait* does not have to be reinvoked. Thus, the client side state automaton of the *query* pattern and its

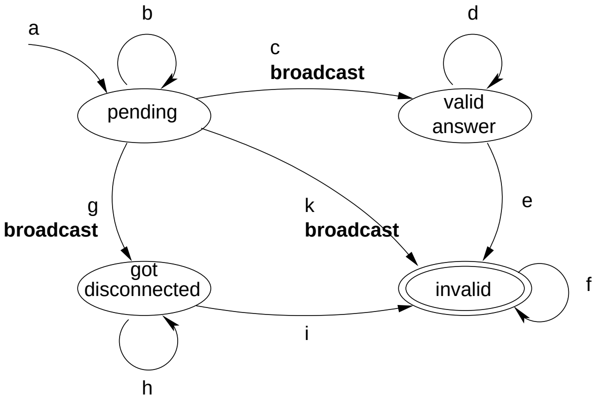


Figure 5.120: *The client side state automaton to manage the lifecycle of a query.*

wait method fully comply with the general policy of the monitors.

Each query is represented by its own monitor instance that is independent of all other open queries. The monitor coordinates everything related to the particular query by its singular state automaton. Each invoked user level method holds its pointer to the corresponding monitor instance. Due to the used smart pointers, the monitor instance disappears only after none of the concurrent calls does need the monitor instance anymore. Thus, each invocation, that holds a pointer to the monitor instance, can get completed and can process a broadcast without getting into troubles due to a disappeared monitor instance. However, only one invocation can consume a valid identifier and all others then correctly experience an invalid identifier due to the meanwhile performed state transitions. Due to the monitor protected state automaton, concurrent calls waiting on the same query identifier neither miss a broadcast nor consume an identifier multiple times. Due to the used smart pointers, even arbitrarily called *discards* do not impose any problems.

The Client Side Administration Internals The client side administration comprises the *add*, the *remove*, the *connect*, the *disconnect* and the *blocking* method. The *add*, *remove* and *connect* methods of the *query* pattern work in the same way as the ones of the *send* pattern. The *blocking* method of the user interface of the *query* client is shown on the left of figure 5.121. It stores the blocking mode in the *user blocking flag* to be able to properly initialize newly generated monitors of newly invoked queries. Furthermore, it forwards every modification of the *user blocking flag* to all the monitors of the *list of monitors*. The *blocking callback* invoked by the component management is shown on the right of figure 5.121. It works in the exactly same way but uses the *component blocking flag* to store the blocking mode and the *blocking component* method instead of the *blocking user* method.

Figure 5.122 shows that part of the *disconnect* method that performs the pattern specific state transitions on the monitors of the service related interactions. Only queries in the *pending* state are affected by a *disconnect* since these cannot get the expected answer anymore. Therefore, the *disconnect* executes the state transition *g* for those so that all methods that currently block on the corresponding monitor get released. Of course, already received responses keep valid and are not affected by a *disconnect*. After the *disconnect* is completed, no more monitors are in the *pending* state and all affected queries are informed properly. The state transitions performed by the *disconnect* are summarized in table 5.49. Since the mutex M1 is hold during the whole *disconnect* procedure and since that also

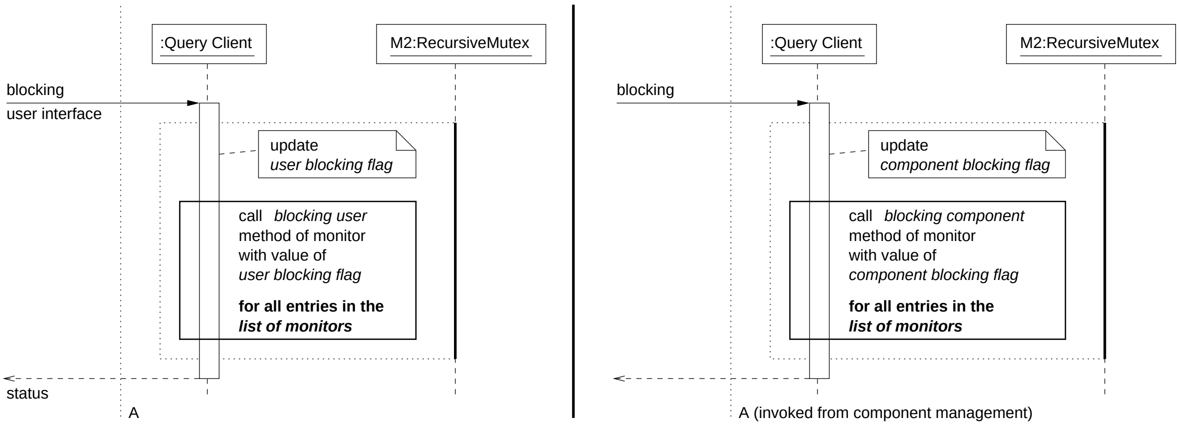


***Figure 5.121:** The internals of the client side* blocking *method.*

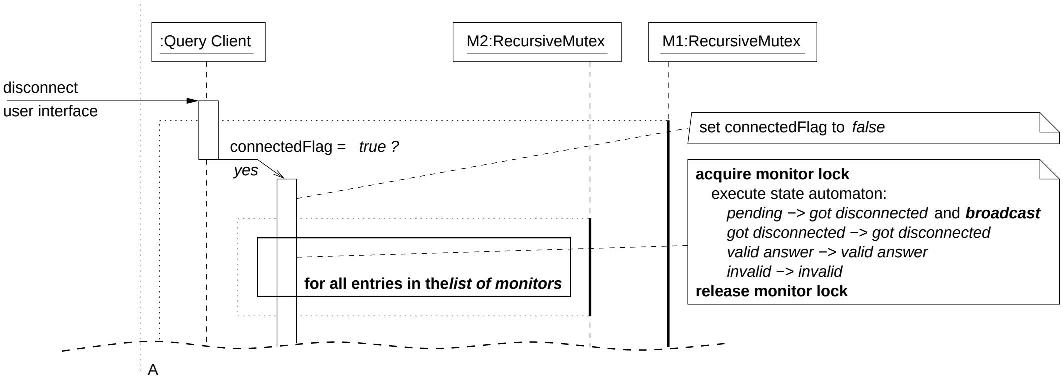


***Figure 5.122:** The internals of the client side* disconnect *method.*

holds true for the *request* method, both cannot interfere so that the *disconnect* cannot miss a newly generated monitor instance.

The Client Side Service Internals The *request*, *receive*, *receive wait* and *discard* methods form the asynchronous user interface that allows the deferred reception of an answer. Figure 5.123 shows the *request* method. A new monitor with a unique query identifier is generated. It is initialized to the *pending* state and is added to the *list of monitors*. In case the request could not be sent successfully, the monitor is set to the *invalid* state and is removed from the *list of monitors* so that it can get destroyed.

The mutex M1 ensures that no disconnect can be executed while a new query is invoked ①. Otherwise, the state automaton of the new monitor instance could miss a state transition from the *pending* to the *got disconnected* state since it is not yet accessible via the *list of monitors*.

The server can send a message that is related to the new query only after it received the message ②. Prior to that point of time, all server side methods return with a *wrong identifier*. Since the message ② is sent only after the monitor got added to the *list of monitors*, any response can already find the monitor and access its state automaton so that no response can be missed even if it arrives just after ②. This also holds true for the server side *discard* since it is based on the regular response.

Automaton state on invocation	New automaton state	Action
pending	*got disconnected*	transition *g*
got disconnected	*got disconnected*	transition *h*
valid answer	*valid answer*	transition *d*
invalid	*invalid*	transition *f*

Table 5.49: *The state transitions performed by the* disconnect *method.*

A client side method can be invoked with an arbitrary query identifier. As soon as the mutex M2 is released at ③, the newly generated monitor is already found in the *list of monitors* so that no *wrong identifier* status is returned anymore. That does not matter since the already invoked methods experience a regular query in case of successfully sending the query request message. In case of an error, the state transition *k* is executed and thus all blocking waits are released for sure. As soon as ④ is reached, there are no more blocking calls on that monitor instance due to its *invalid* state and all methods already return the *wrong identifier* status. As soon as ⑤ is reached, the *wrong identifier* status is returned since the monitor cannot anymore be found in the *list of monitors*. A client side *discard* cannot interfere since that is anyway inhibited by the mutex M1.

The error branch ⑥ is invoked only in case the send of the query request message was not successful. Due to the missing request message, there can be no answer message from the server and thus, only the state transition *k* is required at ⑦. The *got disconnected* state cannot be encountered since a *disconnect* is delayed by the mutex M1. The *valid answer* state is not possible since the request was not sent and the *invalid* state is not possible since a client side *discard* is also delayed by the mutex M1. The *invalid* state can also not be reached via a server side *discard* due to the never sent request message.

Automaton state on invocation	Returned status of answer message	New automaton state	Action
pending	*ok* *error*	*valid data* *invalid*	transition *c*, store received answer transition *k*, discard message
got disconnected	—	*got disconnected*	transition *h*, discard message
valid answer	—	*valid answer*	transition *d*, discard message
invalid	—	*invalid*	transition *f*, discard message

Table 5.50: *The state transitions performed by the callback handler of the* answer *message.*

Figure 5.124 shows the callback handler for incoming answers and table 5.50 summarizes the state transitions performed inside the handler. The monitor belonging to the received answer is non-ambiguously identified by means of the returned client side query identifier. Due to the used *smart pointers*, the mutex M2 can be released as soon as the monitor has been found. In case the answer is expected, the corresponding monitor exists and it is in the *pending* state. Depending on the status returned by the response message, either the transition *c* or *k* is executed. Both transitions include a broadcast so that all blocking calls get released. The returned status is set to *error* either in case the request was rejected at the service provider or in case it was discarded there by the user level *discard*.

A response encounters the *got disconnected* state in case the client invokes a disconnect while an answer is already on its way back to the client. At the client, the state automatons are already set

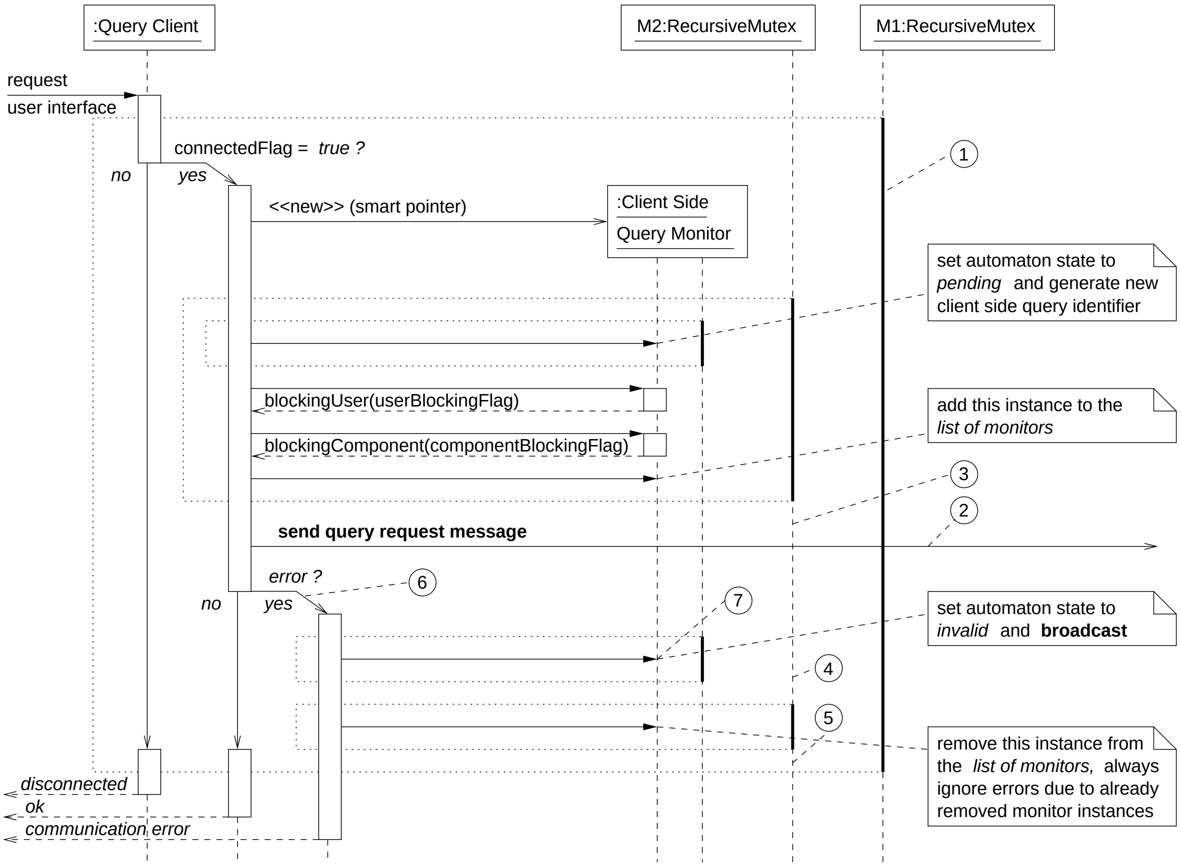

:Query Client
M2:RecursiveMutex
M1:RecursiveMutex
request
user interface
connectedFlag = true ?
no
yes
1
<<new>> (smart pointer)
:Client Side
Query Monitor
set automaton state to pending and generate new client side query identifier
blockingUser(userBlockingFlag)
blockingComponent(componentBlockingFlag)
add this instance to the list of monitors
3
2
send query request message
error ?
6
no
yes
7
set automaton state to invalid and broadcast
4
5
remove this instance from the list of monitors, always ignore errors due to already removed monitor instances
disconnected
ok
communication error

Figure 5.123: *The internals of the client side* request *method.*

appropriately by performing the state transition *g* that already released all blocking calls. The *invalid* state is encountered in case of a client side *discard* and again, all blocking calls are released once the *invalid* state is reached. In case the monitor does even not exist anymore, it is also safe to discard the response since a monitor gets removed only in the *invalid* state and that is reached only after all blocking methods have been signalled. Thus, in all those cases it is safe to discard the response without leaving any pending queries behind.

Figure 5.125 shows the *receive* method and table 5.51 summarizes the state transitions. Since the *receive* method operates only locally without requiring any communication, it does not acquire the mutex M1. It returns *no data* if called in the *pending* state, *disconnected* if called in the *got disconnected* state and it returns the available answer with *ok* if called in the *valid answer* state. Both, the *got disconnected* and the *valid answer* state proceed to the final *invalid* state and consume the identifier. The mutex M2 could be released at ① since once the monitor is found, it does not disappear due to the *smart pointer*. However, the mutex M2 then needs to be reacquired at ②.

Figure 5.126 shows the *receive wait* method and table 5.52 summarizes the state transitions that are performed by the *receive wait* method. The mutex M2 can be released at ① since the monitor cannot get destroyed while the *receive wait* method holds the smart pointer to the monitor instance.

The *wait* method of the monitor blocks only if the blocking condition evaluates to *true* and if the automaton is in the *pending* state. Blocking can be inhibited or aborted by setting the *user blocking*

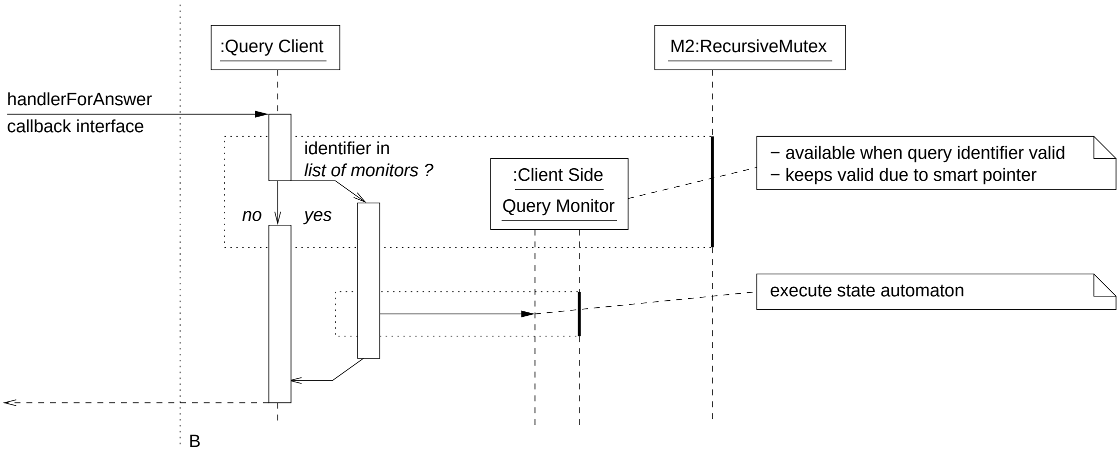


Figure 5.124: *The internals of the client side handler for the* answer *message.*

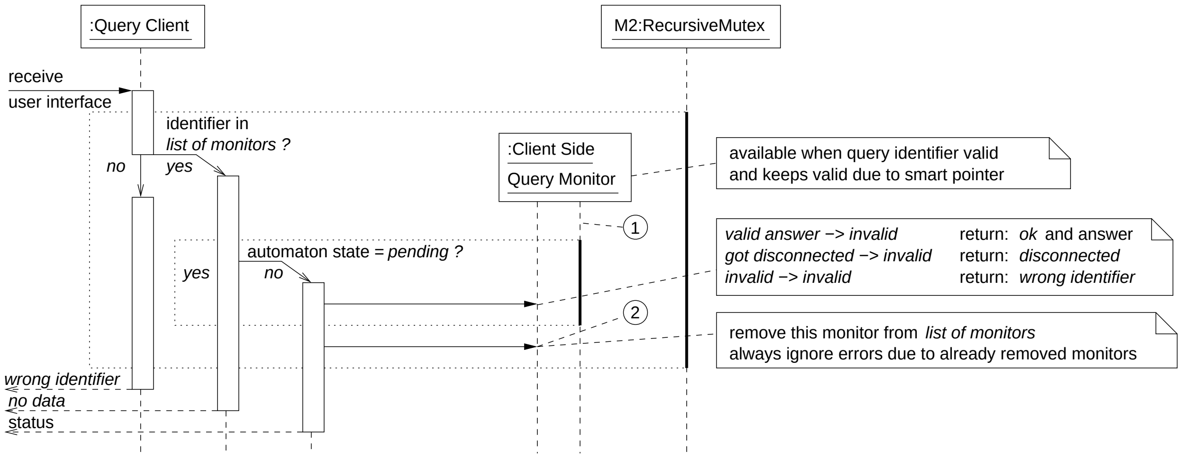


Figure 5.125: *The internals of the client side* receive *method.*

flag or the *component blocking flag* of the monitor to *false*. Invoking the *receive wait* method in the *pending* state with a blocking indicator that evaluates to *true* blocks the method call until either any other state is reached or the blocking mode is set to *false* or both. In all other cases, the *receive wait* method passes the *wait* without getting blocked.

A *pending* state after having passed the *wait* always indicates that blocking is not allowed. Thus, the method call returns with a *cancelled* status. In case of the *got disconnected* state, the state transition *i* is executed, the query identifier is consumed and the *disconnected* status is returned. In case of the *valid answer* state, the state transition *e* is executed, the query identifier is again consumed and the answer is returned with an *ok* status. In case of the *invalid* state, the *wrong identifier* status is returned. The *receive wait* method awakes in the *invalid* state in case the request got discarded by calling the *discard* method, either at the client side or at the server side or in case the request was already rejected inside the server side upcall.

Since the mutex M2 must not be hold when the *wait* of the monitor is invoked, a concurrently

Automaton state on invocation	New automaton state	Action	Returned status
pending	*pending*	transition *b*	*no data*
got disconnected	*invalid*	transition *i*, consume identifier	*disconnected*
valid answer	*invalid*	transition *e*, consume identifier return answer	*ok*
invalid	*invalid*	transition *f*	*wrong identifier*

Table 5.51: *The state transitions performed by the* receive *method.*

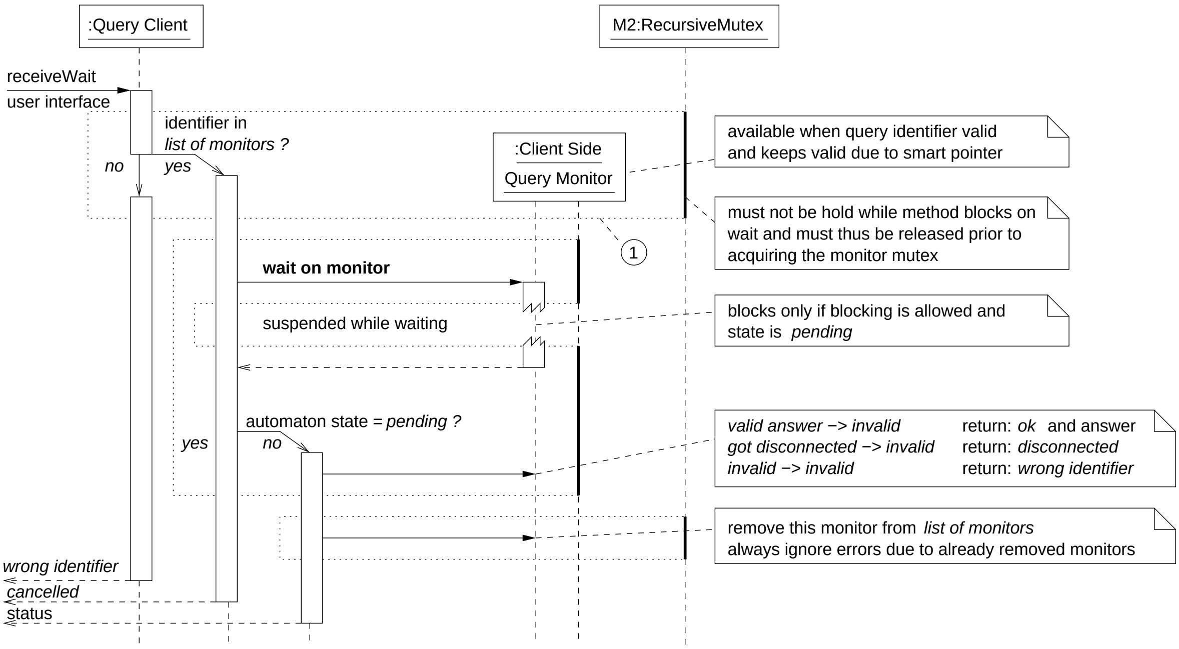

:Query Client
M2:RecursiveMutex
receiveWait
user interface
identifier in
list of monitors ?
no
yes
:Client Side
Query Monitor
available when query identifier valid
and keeps valid due to smart pointer
must not be hold while method blocks on
wait and must thus be released prior to
acquiring the monitor mutex
1
wait on monitor
suspended while waiting
blocks only if blocking is allowed and
state is pending
automaton state = pending ?
yes
no
valid answer −> invalid return: ok and answer
got disconnected −> invalid return: disconnected
invalid −> invalid return: wrong identifier
remove this monitor from list of monitors
always ignore errors due to already removed monitors
wrong identifier
cancelled
status

Figure 5.126: *The internals of the client side* receive wait *method.*

called client side *discard* can remove the monitor from the *list of monitors* so that the monitor can not be found by the upcall of the response. However, a monitor gets removed only in the *invalid* state. Once that state is reached, all blocking *waits* have been signalled and new invocations that just got the monitor prior to its removal from the *list of monitors* do not block anymore due to the *invalid* state. Due to the broadcast, any number of waiting methods always get resumed. Due to the smart pointers, all resumed methods can complete their processing without loosing access to the monitor. Of course, there is always only one method that consumes the query identifier and there happen further state changes so that not every resumed method sees the same state. However, all methods related to a particular query, work on the same state automaton that is coordinated by the monitor so that all methods see a consistent state and return consistent status codes.

Figure 5.127 shows the *discard* method and table 5.53 summarizes the state transitions. Calling the *discard* method invalidates the query identifier by switching to the final *invalid* state. In case the previous automaton state is *pending*, a *request discard* message is sent to the service provider to notify it about the discarded request. Since there can be no *pending* state if the service requestor is not connected to a service provider and since the mutex M1 is not released after saving the previous

Automaton state on invocation	Action
pending	block in *wait* of monitor
got disconnected	rush through *wait*
valid answer	rush through *wait*
invalid	rush through *wait*

State after *wait* is passed	New automaton state	Action	Returned status
pending	*pending*	transition b, do nothing	*cancelled*
got disconnected	*invalid*	transition i, consume identifier	*disconnected*
valid answer	*invalid*	transition e, consume identifier and return answer	*ok*
invalid	*invalid*	transition f, do nothing	*wrong identifier*

Table 5.52: *The state transitions performed by the* receive wait *method.*

Automaton state on invocation	New automaton state	Action	Returned status
pending	*invalid*	transition *k*, consume identifier	*ok*
got disconnected	*invalid*	transition *i*, consume identifier	*ok*
valid answer	*invalid*	transition *e*, consume identifier	*ok*
invalid	*invalid*	transition *f*	*wrong identifier*

Table 5.53: *The state transitions performed by the* discard *method.*

automaton state so that there can no *disconnect* interfere, it would not be necessary to also check the *connected flag* before sending the *request discard* message. In principle, it is not necessary to inform the service provider about a discarded response but that allows the service provider to safe resources by not providing responses that are not needed anymore and just ignored at the service requestor.

The *discard* method can only interfere with a concurrent discard at the service provider. Responses that were already on their way to the service requestor but did not arrive before the client side *discard* method has been invoked are properly rejected by the callback handler for incoming answers. *Request discard* messages that were already on their way to the service provider but did not arrive before either the server side *answer* or *discard* method has been invoked are properly rejected by the callback handler for the *request discard* message.

The *discard* method does not return a *communication error* since that can be ignored. In the worst case, a response not needed anymore gets unnecessarily calculated and returned to the client.

Finally, the synchronous *query* method is shown in figure 5.128. It is simply composed of the already known methods for handling an asynchronous query.

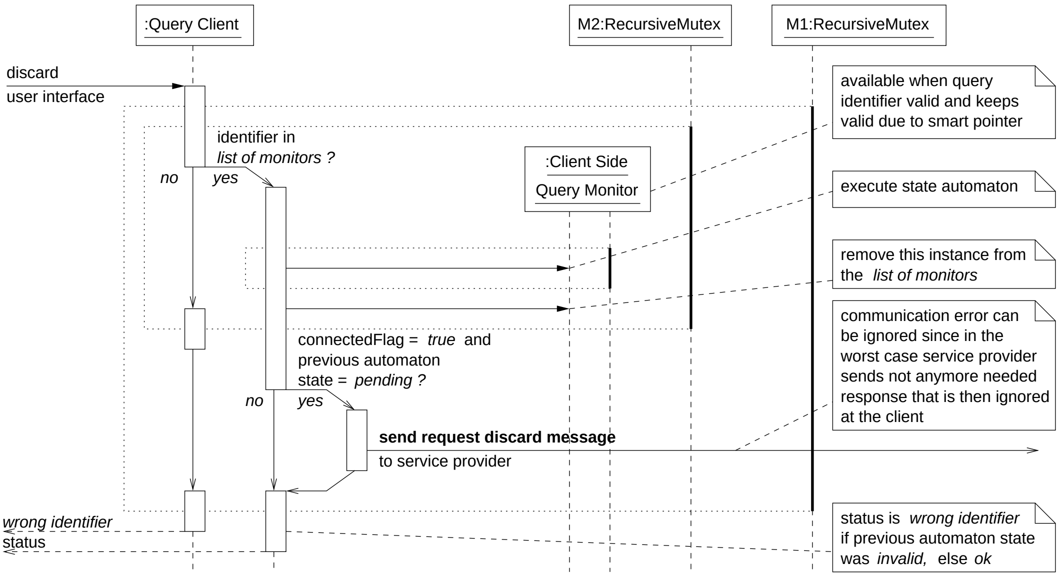

:Query Client
M2:RecursiveMutex
M1:RecursiveMutex
discard
user interface
available when query identifier valid and keeps valid due to smart pointer
identifier in
list of monitors ?
no
yes
:Client Side
Query Monitor
execute state automaton
remove this instance from the list of monitors
communication error can be ignored since in the worst case service provider sends not anymore needed response that is then ignored at the client
connectedFlag = true and
previous automaton
state = pending ?
no
yes
send request discard message
to service provider
wrong identifier
status
status is wrong identifier if previous automaton state was invalid, else ok

Figure 5.127: *The internals of the client side* discard *method.*

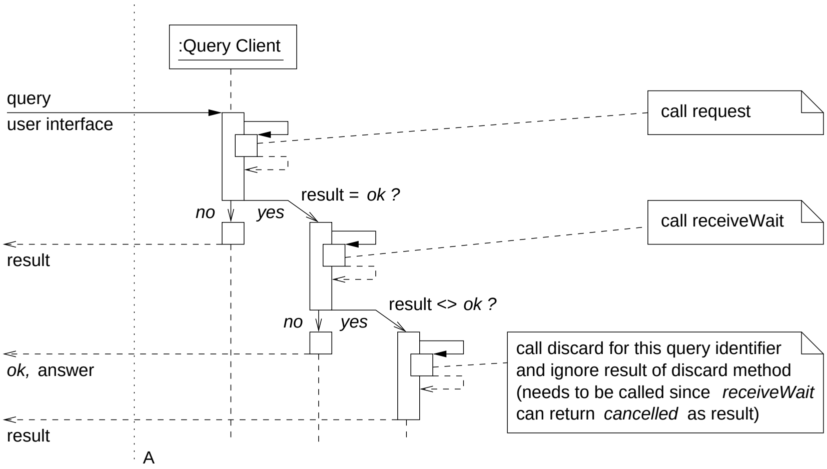
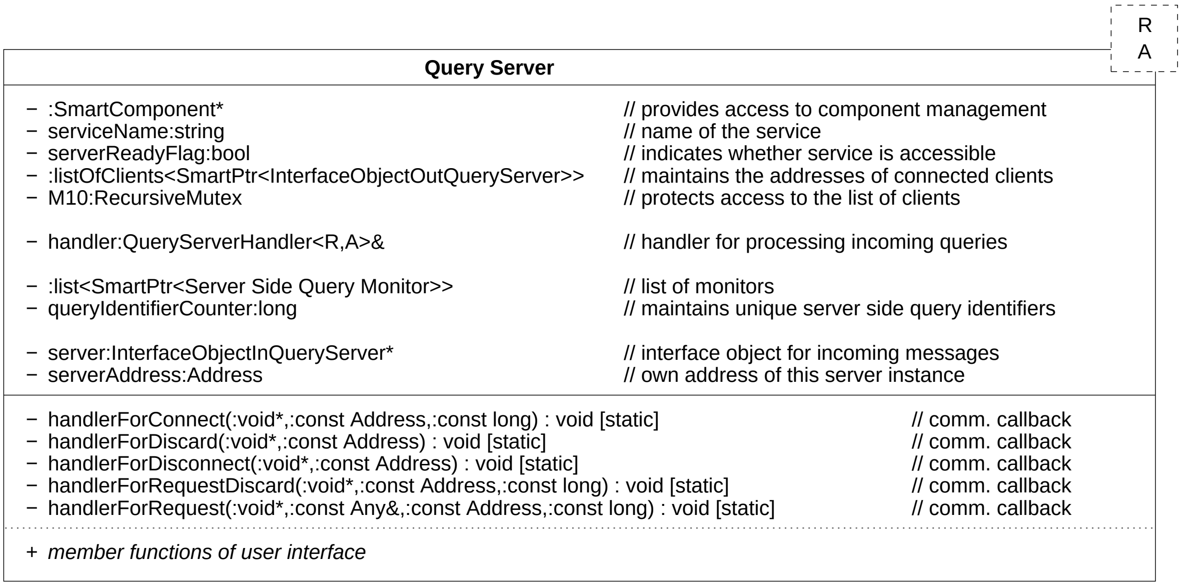

:Query Client
query
user interface
call request
result = ok ?
no
yes
call receiveWait
result
result <> ok ?
no
yes
call discard for this query identifier and ignore result of discard method (needs to be called since receiveWait can return cancelled as result)
ok, answer
result
A

Figure 5.128: *The internals of the client side* query *method.*

The Server Part The server side internals of the *query server* class are shown in figure 5.129. The additional attributes compared to the *send* pattern are explained in table 5.54. The handler for requests accepts the incoming requests. The first argument is the *this*-pointer, the second argument the content of the communication object of the request, the third argument the address of the service requestor that invokes the query and the fourth argument the client side query identifier that has to be returned with the response. The handler for discarding a request accepts the R4 message. Its first argument is the *this*-pointer, the second argument the address of the service requestor and the third argument the client side query identifier. The address and the query identifier allow to identify the request that is to be discarded.



```
Query Server                                                              R
                                                                          A
− :SmartComponent*                                   // provides access to component management
− serviceName:string                                 // name of the service
− serverReadyFlag:bool                               // indicates whether service is accessible
− :listOfClients<SmartPtr<InterfaceObjectOutQueryServer>>   // maintains the addresses of connected clients
− M10:RecursiveMutex                                 // protects access to the list of clients

− handler:QueryServerHandler<R,A>&                   // handler for processing incoming queries

− :list<SmartPtr<Server Side Query Monitor>>         // list of monitors
− queryIdentifierCounter:long                        // maintains unique server side query identifiers

− server:InterfaceObjectInQueryServer*               // interface object for incoming messages
− serverAddress:Address                              // own address of this server instance

− handlerForConnect(:void*,:const Address,:const long) : void [static]                 // comm. callback
− handlerForDiscard(:void*,:const Address) : void [static]                             // comm. callback
− handlerForDisconnect(:void*,:const Address) : void [static]                          // comm. callback
− handlerForRequestDiscard(:void*,:const Address,:const long) : void [static]          // comm. callback
− handlerForRequest(:void*,:const Any&,:const Address,:const long) : void [static]     // comm. callback

+ member functions of user interface
```

Figure 5.129: *Details of the internals of the* query server *class.*

Attribute	Description
:list<SmartPtr<Server Side Query Monitor>>	The *list of monitors* holds the monitors of the currently active queries that are not yet answered. It is protected by the mutex M10.
queryIdentifierCounter:long	The counter provides the unique server side identifier that is assigned to every query. It is protected by the mutex M10.

Table 5.54: *Description of the server part attributes.*

All active queries are represented by their own monitor instance that is shown in figure 5.130. At the *query* server, there are currently no blocking methods so that the monitor is solely used to coordinate the access to its attributes. As long as one does not have to wait for a particular state, a simpler structure would do where one even could use the mutex M10 to protect the automaton from concurrent modifications.

The query identifiers manage the basic version of an asynchronous completion token [138]. A new identifier that is unique within the service requestor is generated with every request and is forwarded to the service provider. At the service provider, every incoming request gets another identifier which

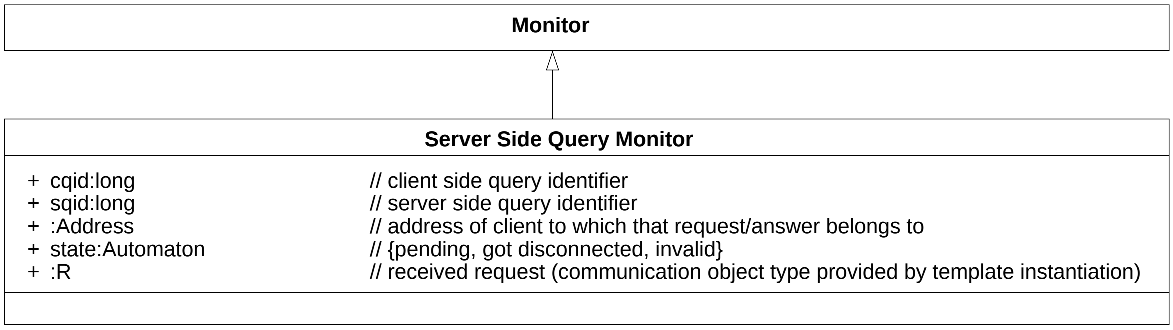

Monitor
Server Side Query Monitor
+ cqid:long // client side query identifier
+ sqid:long // server side query identifier
+ :Address // address of client to which that request/answer belongs to
+ state:Automaton // {pending, got disconnected, invalid}
+ :R // received request (communication object type provided by template instantiation)

Figure 5.130: *The server side representation of an active query.*

is unique within the service provider. This is necessary since requests of several service requestors can be labeled with the same client side identifier. Thus, the client provided identifiers are unique only in combination with the client address. The server side identifier is forwarded to the request handler and has to be returned with the answer. It now allows to identify to which open request the answer belongs to and indicates to which service requestor the answer has to be sent to. The answer message contains both the answer itself and the before received client side identifier. The client side identifier then allows to assign the received answer to the proper open request.

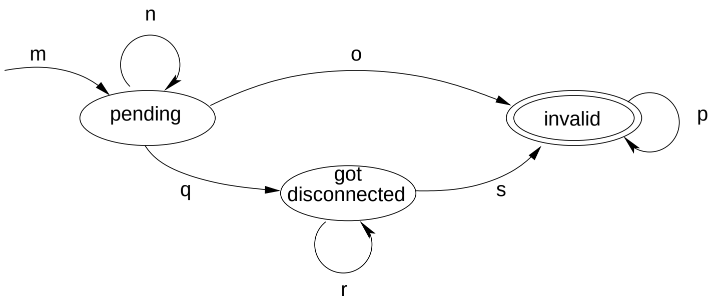


Figure 5.131: *The server side state automaton to manage the lifecycle of a query.*

Figure 5.131 shows the server side state automaton of a query. A new request initializes its state automaton to the *pending* state which indicates that the answer still has to be provided. Providing the answer in the *pending* state performs the state transition *o* to the final *invalid* state in which the monitor is removed from the *list of monitors* which invalidates the server side query identifier. If a client gets disconnected, all its open queries are set to the *got disconnected* state at the server by executing the state transition *q*. This prevents the server from sending back answers to already disconnected clients or even to clients not existing anymore. Thus, answers that are provided in either the *got disconnected* or the *invalid* state are ignored. Providing the answer or invoking the *check* method in the *got disconnected* state performs the state transition *s* and thus consumes the query identifier. The query identifier is also always consumed by a *discard*, that either executes the state transition *o* or *s*, independently of whether the discard results from the client side or the server side *discard* method.

The Server Side Administration Internals The server part of the *connect* and the *server initiated disconnect* comply with the generic form of those administrative interactions. The relevant part of the handler for the *disconnect* message is shown in figure 5.132. It receives the client address and iterates through the *list of monitors* to perform the transition *q* for all open queries of that client. The mutex M10 inhibits any methods that need to interact with a connected client while the *disconnect* is active.

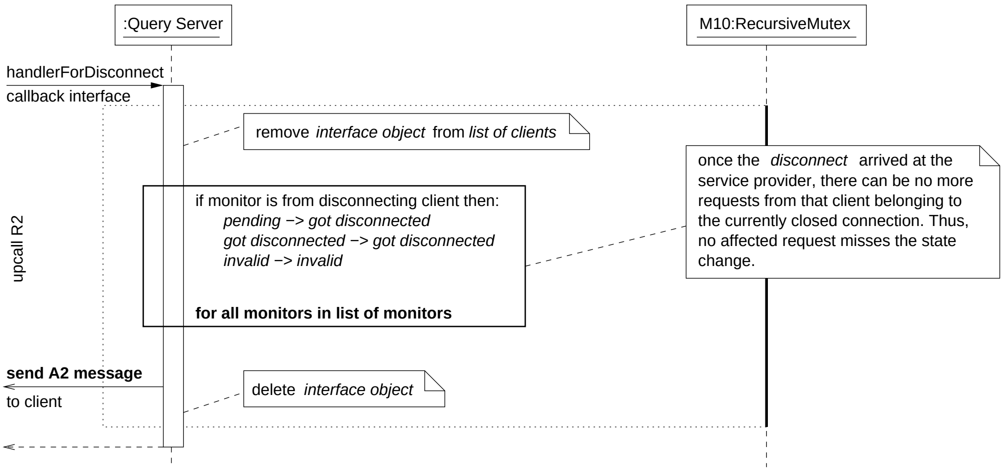

:Query Server
M10:RecursiveMutex
handlerForDisconnect
callback interface
upcall R2
remove *interface object* from *list of clients*
if monitor is from disconnecting client then:
pending -> got disconnected
got disconnected -> got disconnected
invalid -> invalid
for all monitors in list of monitors
once the *disconnect* arrived at the service provider, there can be no more requests from that client belonging to the currently closed connection. Thus, no affected request misses the state change.
send A2 message
to client
delete *interface object*

Figure 5.132: *The internals of the server side handler for the* disconnect *message.*

The Server Side Service Internals Figure 5.133 shows the handler for incoming requests. The handler creates a *server side query monitor* that is added to the *list of monitors*. In case that something went wrong either when creating the monitor or when adding it to the *list of monitors*, one cannot simply drop the request since the client has to know that it cannot expect an answer. Thus, the handler sends the *answer* message with an *error* status to indicate that the contained communication object is not valid. In case that nothing went wrong, the user provided handler is invoked to process the request.

The *answer* method is shown in figure 5.134. The answer is sent only in case the corresponding monitor is in the *pending* state. The *got disconnected* and the *invalid* state indicate that the provided answer is not needed anymore. Thus, the provided response just gets discarded and no response message is sent. At the service provider, the *got disconnected* and *invalid* states are reached only after either a response or a discard has been sent so that there are no more pending method calls at the client or after the client already released all its blocking calls due to a disconnect or a discard. In all those cases, the not sent message does not leave any pending calls behind. The query identifier gets consumed by the state transitions *o* and *s*.

Since the mutex M10 is hold for the whole *answer* method, there can be no interfering disconnect. Thus, one does not have to check whether the client is still accessible before sending the response. The *pending* state is sufficient to ensure the availability of the corresponding service requestor.

The *check* method is shown in figure 5.135. Due to the handler based interface there is no simple way to abort the processing of responses not needed anymore. The *check* method allows to check whether the response is still needed. Regular checks are in particular reasonable with resource intensive request processings. Again, the query identifier is consumed by the state transition *s*.

Figure 5.136 shows the internals of *discard* method. In case it gets invoked on a query in the *pending* state, it sends the *answer* message but with the status set to *error* to indicate that the response does not contain a valid communication object. The handler for incoming responses at the service

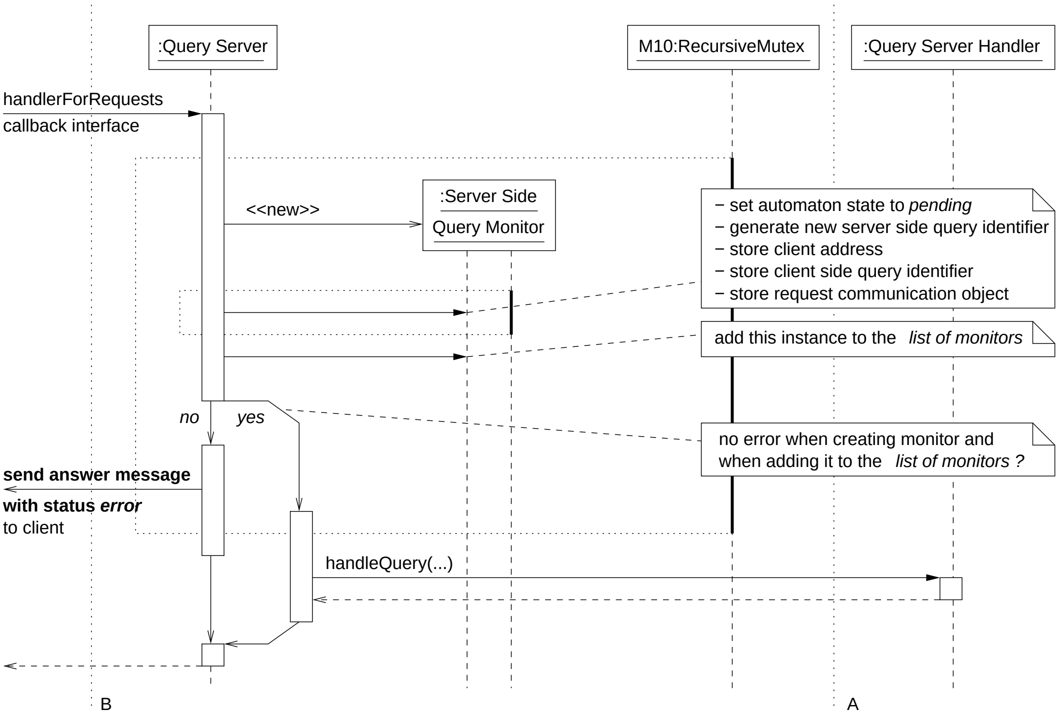

:Query Server
M10:RecursiveMutex
:Query Server Handler
handlerForRequests
callback interface
<<new>>
:Server Side
Query Monitor
− set automaton state to *pending*
− generate new server side query identifier
− store client address
− store client side query identifier
− store request communication object
add this instance to the *list of monitors*
no
yes
no error when creating monitor and
when adding it to the *list of monitors ?*
send answer message
with status *error*
to client
handleQuery(...)
B
A

Figure 5.133: *The internals of the server side* request *handler.*

requestor can then appropriately set the state automaton of the corresponding monitor. The query identifier gets consumed by the state transitions *o* and *s*. The mutex M10 is applied in the same way as with the *answer* method.

Figure 5.137 illustrates the internals of the *request discard* handler. It sets the state of the corresponding monitor to *invalid* and removes the monitor from the *list of monitors*. Furtheron, all methods return a *wrong identifier* status. The server side *answer* method, the server side *discard* method and the client side *discard* method never interfere with each other. At the client, in the worst case, a not anymore needed response message with either an *ok* or an *error* status arrives. That gets properly discarded either due to a not anymore existing monitor or due to the monitor's *invalid* state. At the server, in the worst case, a *request discard* message arrives for a not anymore existing request. It also gets properly discarded either due to the not anymore existing monitor or due to its *invalid* state.

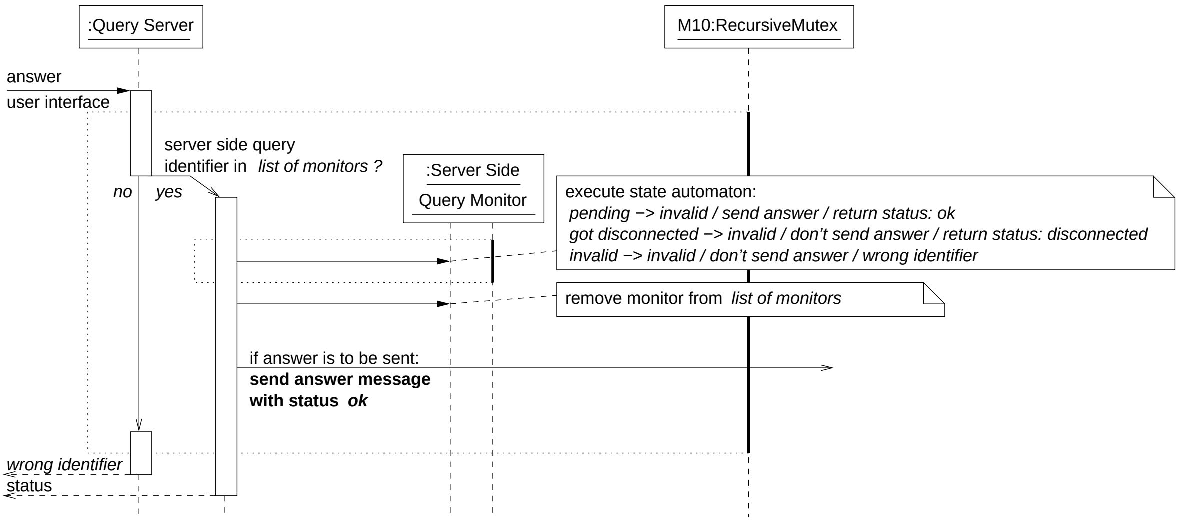

:Query Server
M10:RecursiveMutex
answer
user interface
server side query
identifier in list of monitors ?
no
yes
:Server Side
Query Monitor
execute state automaton:
pending −> invalid / send answer / return status: ok
got disconnected −> invalid / don't send answer / return status: disconnected
invalid −> invalid / don't send answer / wrong identifier
remove monitor from list of monitors
if answer is to be sent:
send answer message
with status ok
wrong identifier
status

Figure 5.134: *The internals of the server side* answer *method.*

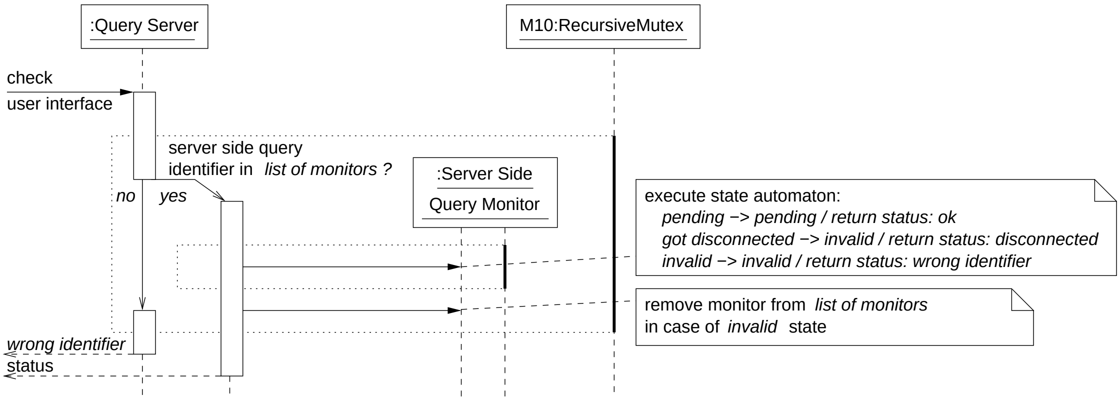

:Query Server
M10:RecursiveMutex
check
user interface
server side query
identifier in list of monitors ?
no
yes
:Server Side
Query Monitor
execute state automaton:
pending −> pending / return status: ok
got disconnected −> invalid / return status: disconnected
invalid −> invalid / return status: wrong identifier
remove monitor from list of monitors
in case of invalid state
wrong identifier
status

Figure 5.135: *The internals of the server side* check *method.*

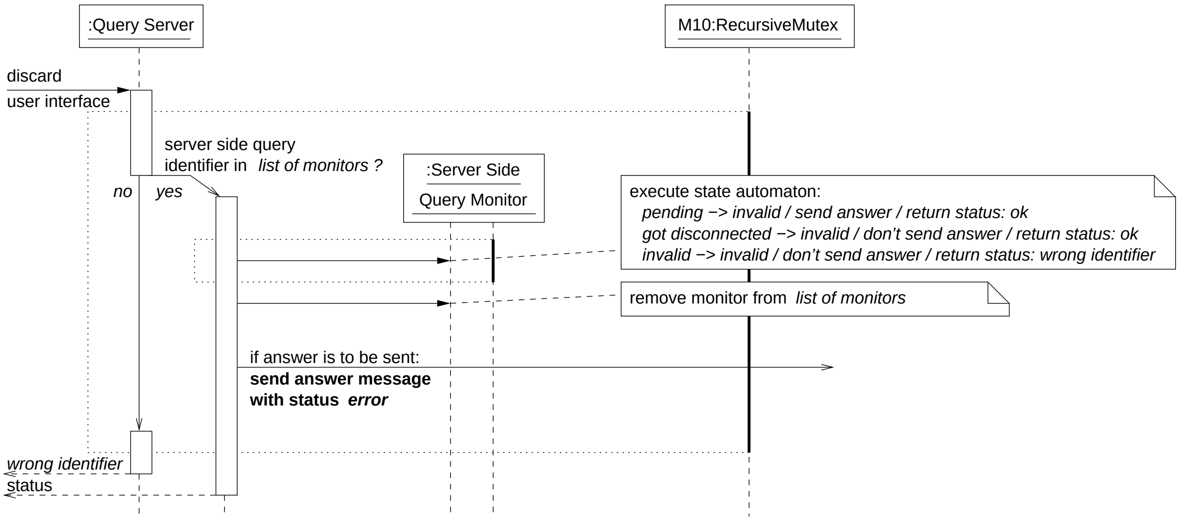

:Query Server
M10:RecursiveMutex
discard
user interface
server side query
identifier in list of monitors ?
no
yes
:Server Side
Query Monitor
execute state automaton:
pending −> invalid / send answer / return status: ok
got disconnected −> invalid / don't send answer / return status: ok
invalid −> invalid / don't send answer / return status: wrong identifier
remove monitor from list of monitors
if answer is to be sent:
send answer message
with status error
wrong identifier
status

Figure 5.136: *The internals of the server side* discard *method.*

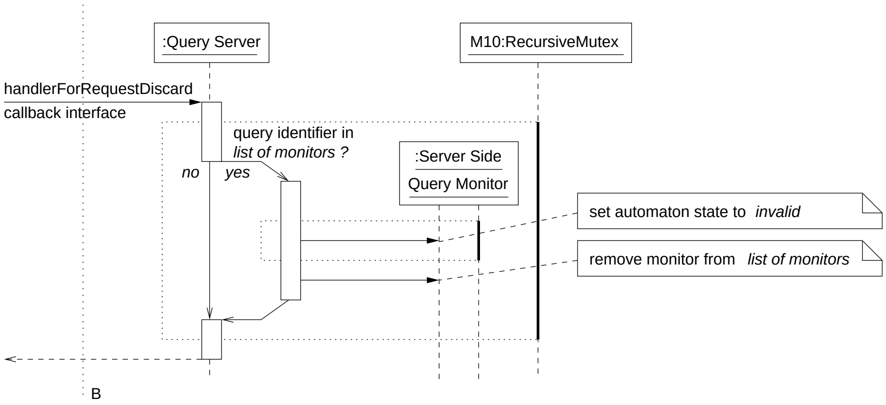
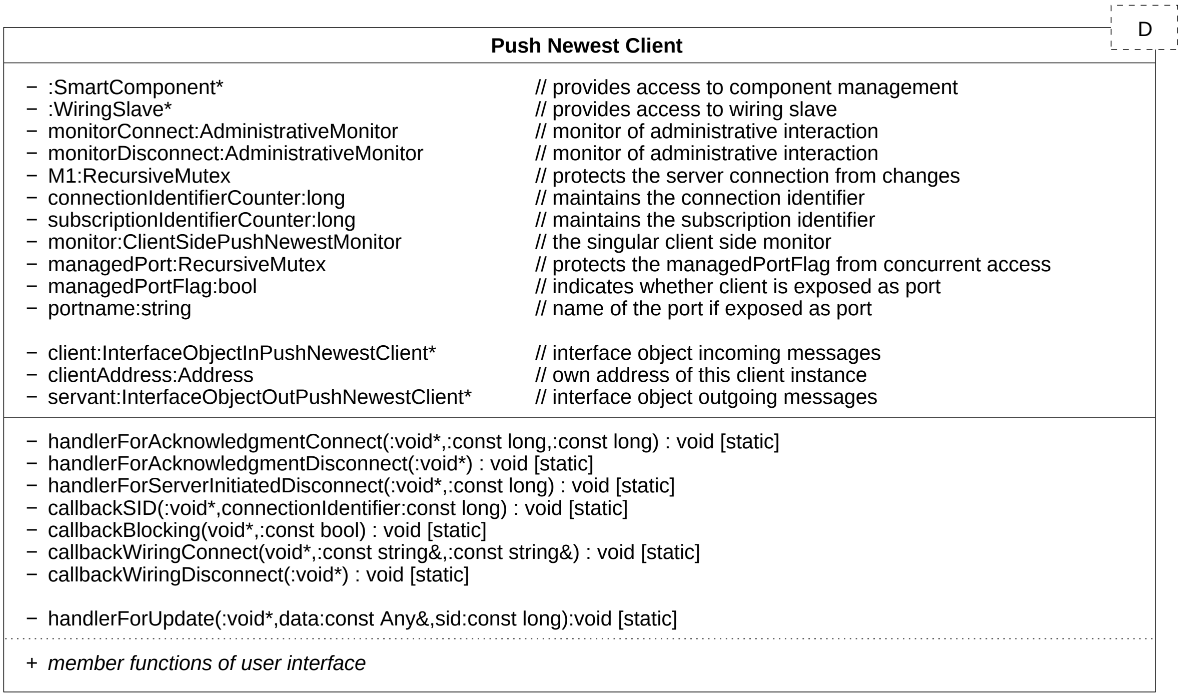


Figure 5.137: *The internals of the server side* request discard *handler.*

5.6.7.4 The Push Newest Pattern

The internals of the *push* patterns are easily graspable once one is familiar with the internals of the *send* and the *query* pattern.

The Client Part The client side internals of the *push newest client* class are shown in figure 5.138. The pattern specific attributes are summarized in table 5.55. Since the monitor exists as long as the client instance exists, one can handle both the connection status and the subscription status via the state automaton inside the monitor and one does not need to buffer the blocking modes. Thus, the *push* client does neither require the mutex M2, the *list of monitors*, the *connected flag*, a *subscribed flag*, the *user blocking flag* nor the *component blocking flag*. Since there is only one monitor, one does not need to search for it but can access it directly.



```
− :SmartComponent*                                 // provides access to component management
− :WiringSlave*                                    // provides access to wiring slave
− monitorConnect:AdministrativeMonitor             // monitor of administrative interaction
− monitorDisconnect:AdministrativeMonitor          // monitor of administrative interaction
− M1:RecursiveMutex                                // protects the server connection from changes
− connectionIdentifierCounter:long                 // maintains the connection identifier
− subscriptionIdentifierCounter:long               // maintains the subscription identifier
− monitor:ClientSidePushNewestMonitor              // the singular client side monitor
− managedPort:RecursiveMutex                       // protects the managedPortFlag from concurrent access
− managedPortFlag:bool                             // indicates whether client is exposed as port
− portname:string                                  // name of the port if exposed as port

− client:InterfaceObjectInPushNewestClient*        // interface object incoming messages
− clientAddress:Address                            // own address of this client instance
− servant:InterfaceObjectOutPushNewestClient*      // interface object outgoing messages

− handlerForAcknowledgmentConnect(:void*,:const long,:const long) : void [static]
− handlerForAcknowledgmentDisconnect(:void*) : void [static]
− handlerForServerInitiatedDisconnect(:void*,:const long) : void [static]
− callbackSID(:void*,connectionIdentifier:const long) : void [static]
− callbackBlocking(void*,:const bool) : void [static]
− callbackWiringConnect(void*,:const string&,:const string&) : void [static]
− callbackWiringDisconnect(:void*) : void [static]

− handlerForUpdate(:void*,data:const Any&,sid:const long):void [static]

+ member functions of user interface
```

Figure 5.138: *Details of the internals of the* push newest client *class.*

Attribute	Description
monitor:ClientSidePushNewestMonitor	The *push* client requires a singular monitor only since it provides the same data to all method invocations. Thus, neither a *list of monitors* nor the mutex M2 is needed. Since the monitor exists as long as the client instance exists, no *smart pointer* is needed.
subscriptionIdentifierCounter:long	That counter provides the unique identifier that is assigned to every *subscribed* session. It allows to identify outdated update messages and is protected by the mutex M1 as the *connectionIdentifierCounter*.

Table 5.55: *Description of the client part attributes.*

The subscription cannot fail since the required administrative structures are already set up with the *connect*. The first argument of the handler for the incoming updates is the *this*-pointer, the second argument the content of the used communication object and the third argument the client side subscription identifier that was provided with the subscription.

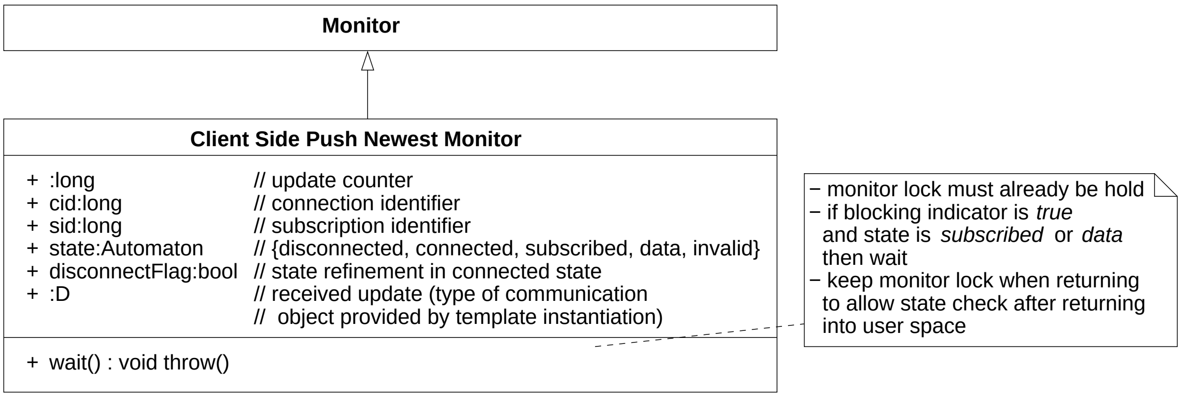

Monitor
Client Side Push Newest Monitor
+ :long // update counter
+ cid:long // connection identifier
+ sid:long // subscription identifier
+ state:Automaton // {disconnected, connected, subscribed, data, invalid}
+ disconnectFlag:bool // state refinement in connected state
+ :D // received update (type of communication
// object provided by template instantiation)
+ wait() : void throw()
− monitor lock must already be hold
− if blocking indicator is *true* and state is *subscribed* or *data* then wait
− keep monitor lock when returning to allow state check after returning into user space

***Figure 5.139:** The client side monitor of the* push newest *pattern.*

Figure 5.139 shows the client side monitor of the *push newest* pattern. The update counter is used by the *getUpdateWait* method. The *wait* method of the monitor blocks only if the state of the automaton is either *subscribed* or *data* since these are the only states in which another update can be expected. Of course, the blocking indicator must also evaluate to *true*. The *disconnect flag* provides additional information in case of the *connected* state as explained below.

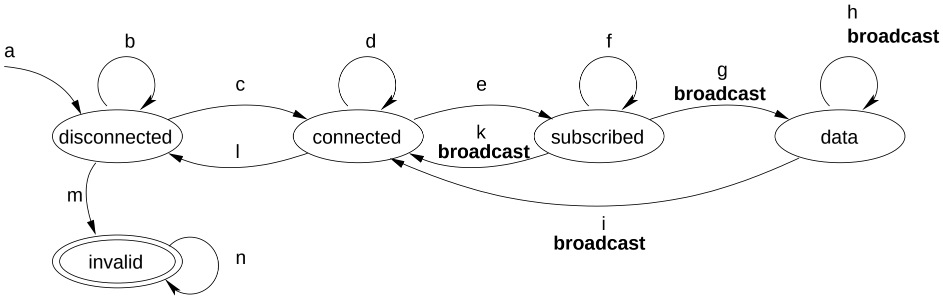


***Figure 5.140:** The client side state automaton to coordinate the* push newest *client.*

The client side state automaton is shown in figure 5.140. The *invalid* state is set prior to destroying the client. In the *disconnected* state, the client is not connected to a service provider and thus, it is also not subscribed and there are no valid updates available. In the *connected* state, it is connected to a service provider but not yet subscribed and thus, there are again no valid updates available. In the *subscribed* state, the client is connected and subscribed to a service provider and awaits its first update and the *data* state indicates that at least one valid update has been received since getting subscribed.

The state transition *c* is performed by a *connect*, the transition *e* by a *subscribe*, the transitions *i* and *k* by an *unsubscribe* and the transition *l* by a *disconnect*. A *disconnect* always first executes an *unsubscribe*. The transitions *g* and *h* are performed with a received update.

The *data* state is the only state in which the *getUpdate* method returns a valid update. The *getUpdateWait* method waits for the next update. Thus, it blocks in the *subscribed* state to get unblocked with the first update and it blocks in the *data* state to get unblocked with the next update. Since the *wait* of the monitor blocks in the *subscribed* and the *data* state, the transitions *g*, *i* and *k* have to broadcast a signal. The transition *h* broadcasts a signal since an update remains in the *data* state but the *getUpdateWait* has to be informed about a newly arrived update. Again, one has to check that a broadcast never releases a blocking method such that it would have to reinvoke the *wait*. In case of the transitions *g* and *h*, an update arrived and returning from the method call is the desired behavior. In case of the transitions *i* and *k*, an *unsubscribe* occurred. Again, the desired behavior of the *getUpdateWait* method is to return since from now on no more updates arrive that belong to the subscription phase in which the *getUpdateWait* method was invoked. Thus, the client side state automaton of the *push newest* pattern and the *wait* method of the monitor fully comply with the general policy of the monitors.

The Client Side Administration Internals The *add*, *remove*, *connect* and *disconnect* methods work like their equivalents at the other patterns. Of course, a *disconnect* first performs an *unsubscribe* prior to invoking the standard disconnect procedure. Thus, before invoking the *unsubscribe*, it sets the *disconnect flag* to *true* so that the blocking member functions that get released due to the *unsubscribe* can detect that the *unsubscribe* is part of a *disconnect*. That is important to return the appropriate status that indicates the reason for getting aborted. The mutex M1 must not be released in between since otherwise another *subscribe* could interfere. Likewise, the *subscribe* method always first invokes an *unsubscribe*. A *connect* updates the *connection identifier* inside the monitor and sets the *disconnect flag* to *false*. A *subscribe* updates the *subscription identifier*. The *subscribe* and the *unsubscribe* methods are shown in figure 5.141.

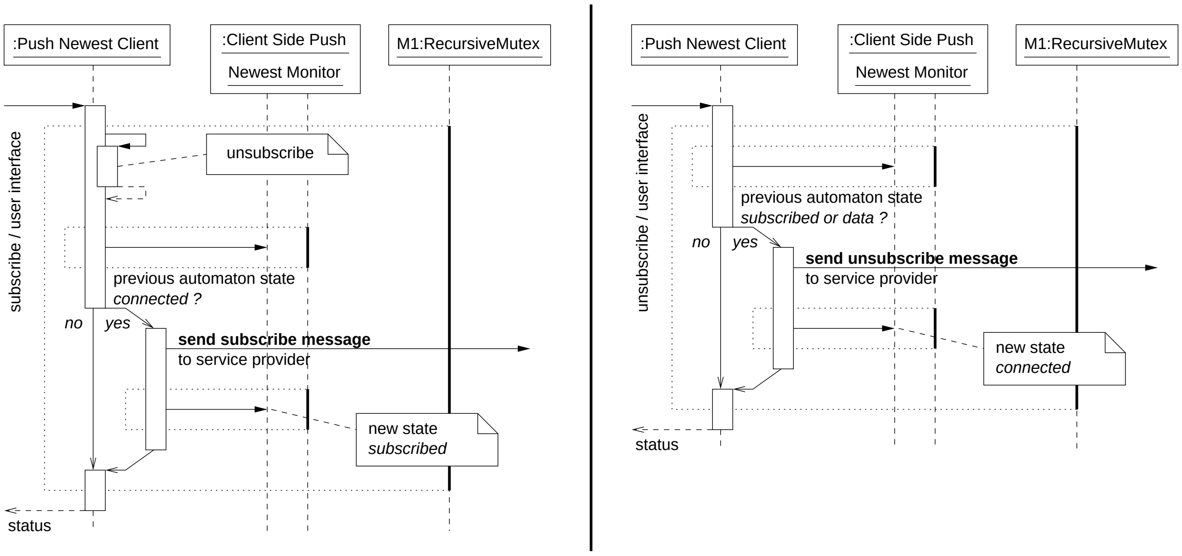


Figure 5.141: *The internals of the client side* subscribe *and* unsubscribe *methods.*

The Client Side Service Internals Figure 5.142 illustrates the client side processing of the *update* message. In case the update message contains a valid subscription identifier, the received update is stored in the monitor where it overwrites previously received updates. Furthermore, a signal is broadcasted. In all other cases, the received message is discarded. Thus, in the *connected*, *disconnected* and *invalid* state, outdated messages are recognized since there can be no relevant update message if the client is not subscribed. In the *subscribed* and the *data* state, the *sid* represents a valid subscription identifier so that one can identify outdated messages. These can occur in case one unsubscribes and subscribes in quick succession.

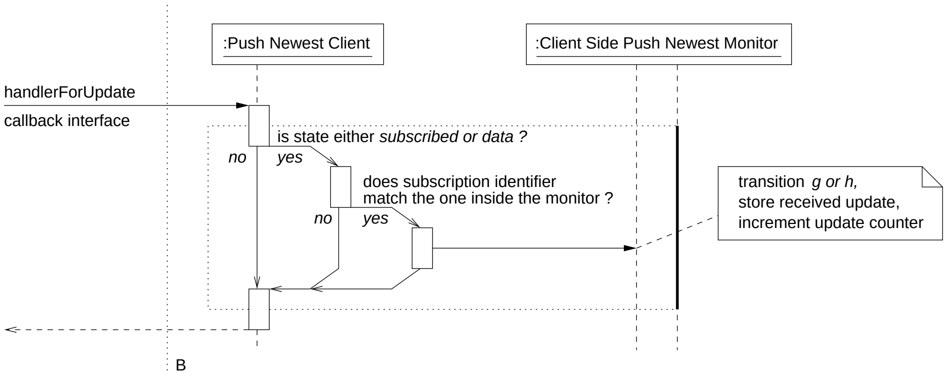

:Push Newest Client
:Client Side Push Newest Monitor
handlerForUpdate
callback interface
is state either *subscribed or data ?*
no
yes
does subscription identifier
match the one inside the monitor ?
no
yes
transition *g or h,*
store received update,
increment update counter
B

Figure 5.142: *The internals of the client side handler for the update message.*

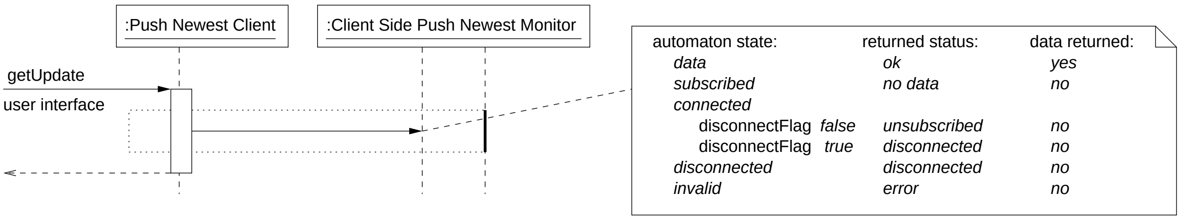

:Push Newest Client
:Client Side Push Newest Monitor
getUpdate
user interface
automaton state:
returned status:
data returned:
data *ok* *yes*
subscribed *no data* *no*
connected
disconnectFlag *false* *unsubscribed* *no*
disconnectFlag *true* *disconnected* *no*
disconnected *disconnected* *no*
invalid *error* *no*

Figure 5.143: *The internals of the client side* getUpdate *method.*

The *getUpdate* method is shown in figure 5.143. It does not perform any state transition but checks the state to decide on the result of the method invocation.

Automaton state on invocation	Action
data	block in *wait* of monitor
subscribed	block in *wait* of monitor
connected	rush through *wait*
disconnected	rush through *wait*
invalid	rush through *wait*

Table 5.56: *The blocking behavior of the* getUpdateWait *method.*

The *getUpdateWait* method is shown in figure 5.144 and the blocking behavior is summarized in table 5.56. The *wait* method of the monitor blocks only if the blocking condition evaluates to *true* and if the automaton is either in the *subscribed* or *data* state. Invoking the *getUpdateWait* method in

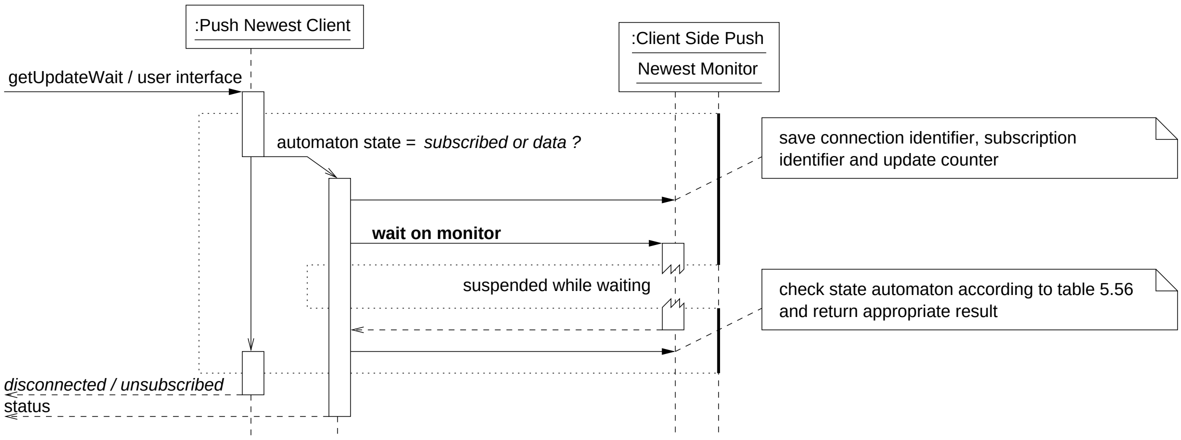

:Push Newest Client
:Client Side Push Newest Monitor
getUpdateWait / user interface
automaton state = subscribed or data ?
save connection identifier, subscription identifier and update counter
wait on monitor
suspended while waiting
check state automaton according to table 5.56 and return appropriate result
disconnected / unsubscribed
status

Figure 5.144: *The internals of the client side* getUpdateWait *method.*

the *subscribed* or the *data* state with a blocking indicator that evaluates to *true* blocks the method call until any other state is reached or until the *data* state is reached again via the transition *h* due to an update or until the blocking mode is set to *false*. In all other cases, the *getUpdateWait* method passes the *wait* of the monitor without getting blocked.

Since one does not generate another monitor with every subscription, one has to carefully check the situation after having passed the *wait*. Due to the Mesa-style semantics of the condition variables of the monitors, there can be further state transitions between getting the signal and resuming the *wait*. In the worst case, there cannot only be an *unsubscribe* and another *subscribe* but even another *connect* to a completely different service provider that might even already sent an update. A *getUpdateWait* must never return an update that belongs to a different session than the one it was invoked in. It is not a problem to return a newer update than that one that released the *wait* as long as the update belongs to the same session. Thus, the connection identifier *cid*, the subscription identifier *sid* and the update counter are saved prior to invoking the *wait* of the monitor to be able to compare the saved values with the current ones after the *wait* was passed. The various combinations and their effects are summarized in table 5.57. An *x* means that one does not have to care about the value. Due to the *cid*, the *sid* and the update counter, each invocation of the *getUpdateWait* method can individually detect the reason which allowed it to pass the *wait* even though all concurrent invocations work on the same monitor instance. Thus, the *getUpdateWait* method never returns a wrong update even if there are further state transitions prior to getting resumed after a signal.

The Server Part The server side internals of the *push newest server* class are shown in figure 5.145. It solely contains already known attributes. Since the server only needs to know whether a connected client is subscribed and since all subscribed clients get the same updates without requiring the maintenance of any individual states and since there are no methods waiting on a particular state, advanced monitors would be wasted within the *push newest* server.

With every connected client, the server needs to hold a *subscribed flag* that indicates whether that client is subscribed and additionally the client provided subscription identifier *csid* in case of being subscribed. Normally, these attributes are stored in a monitor and accessing them is coordinated by the monitor mutex. However, both a monitor and such a fine-grained locking is wasted with the *push newest* server since those attributes are mostly accessed in combination with methods that anyway

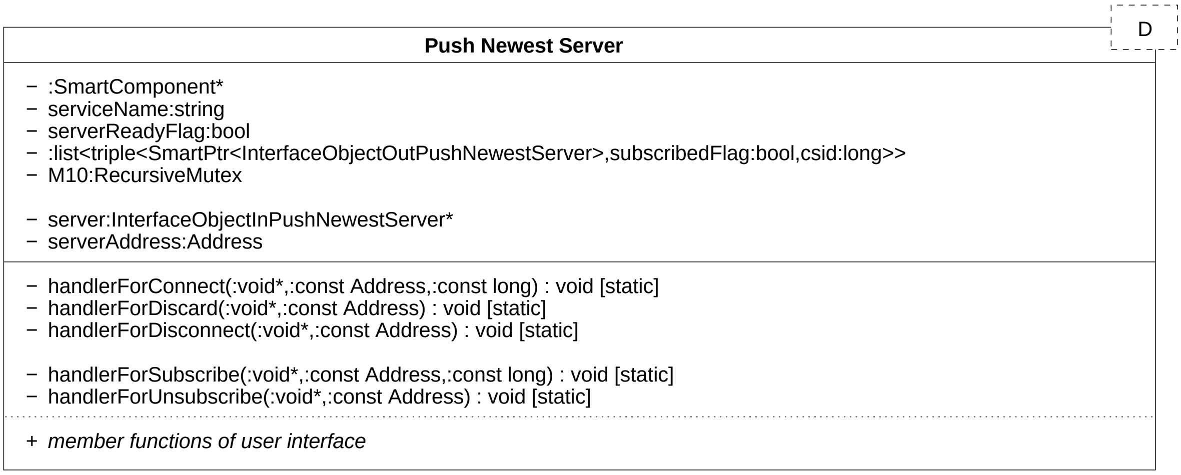

D
Push Newest Server
− :SmartComponent*
− serviceName:string
− serverReadyFlag:bool
− :list<triple<SmartPtr<InterfaceObjectOutPushNewestServer>,subscribedFlag:bool,csid:long>>
− M10:RecursiveMutex
− server:InterfaceObjectInPushNewestServer*
− serverAddress:Address
− handlerForConnect(:void*,:const Address,:const long) : void [static]
− handlerForDiscard(:void*,:const Address) : void [static]
− handlerForDisconnect(:void*,:const Address) : void [static]
− handlerForSubscribe(:void*,:const Address,:const long) : void [static]
− handlerForUnsubscribe(:void*,:const Address) : void [static]
+ member functions of user interface

Figure 5.145: *Details of the internals of the* push newest server *class.*

acquire the mutex M10. Since these mostly iterate through all connected clients, one cannot take advantage from a more fine-grained locking so that it makes no difference to right away use the mutex M10. It is important to remember, that the server side upcalls are allowed to acquire the mutex M10 and that the upcalls can thus access these attributes. The advantage of using the mutex M10 is that one can now extend the *list of clients* to hold the additional attributes since that list is already protected by the mutex M10. As shown in figure 5.145, the *list of clients* contains triples to also hold the *subscribed flag* and the *csid* besides the interface objects.

The Server Side Administration Internals The server parts of the *connect*, the *disconnect* and the *server initiated disconnect* of the *push newest* server comply with the generic form of those administrative interactions. All parts that are related to the *list of monitors* are removed. The handlers to execute a *subscribe* and an *unsubscribe* are shown in figure 5.146.

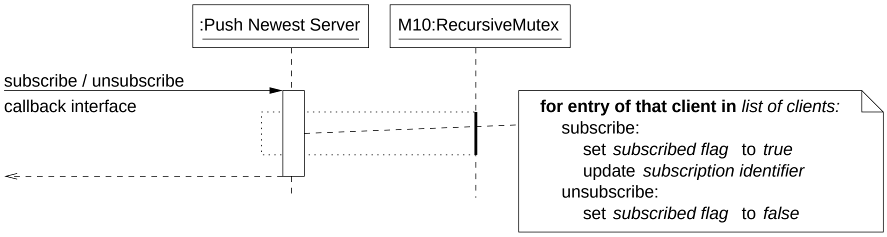

:Push Newest Server
M10:RecursiveMutex
subscribe / unsubscribe
callback interface
for entry of that client in list of clients:
subscribe:
set subscribed flag to true
update subscription identifier
unsubscribe:
set subscribed flag to false

Figure 5.146: *The internals of the server side* subscribe *and* unsubscribe *handlers.*

The Server Side Service Internals The server side user interface consists of the *put* method. It acquires the mutex M10 and iterates through the *list of clients* to send the update message to those clients where the *subscribed flag* is set to *true*. The subscription identifier is individually set to the value that was provided with the client's subscription. Finally, it releases the mutex M10.

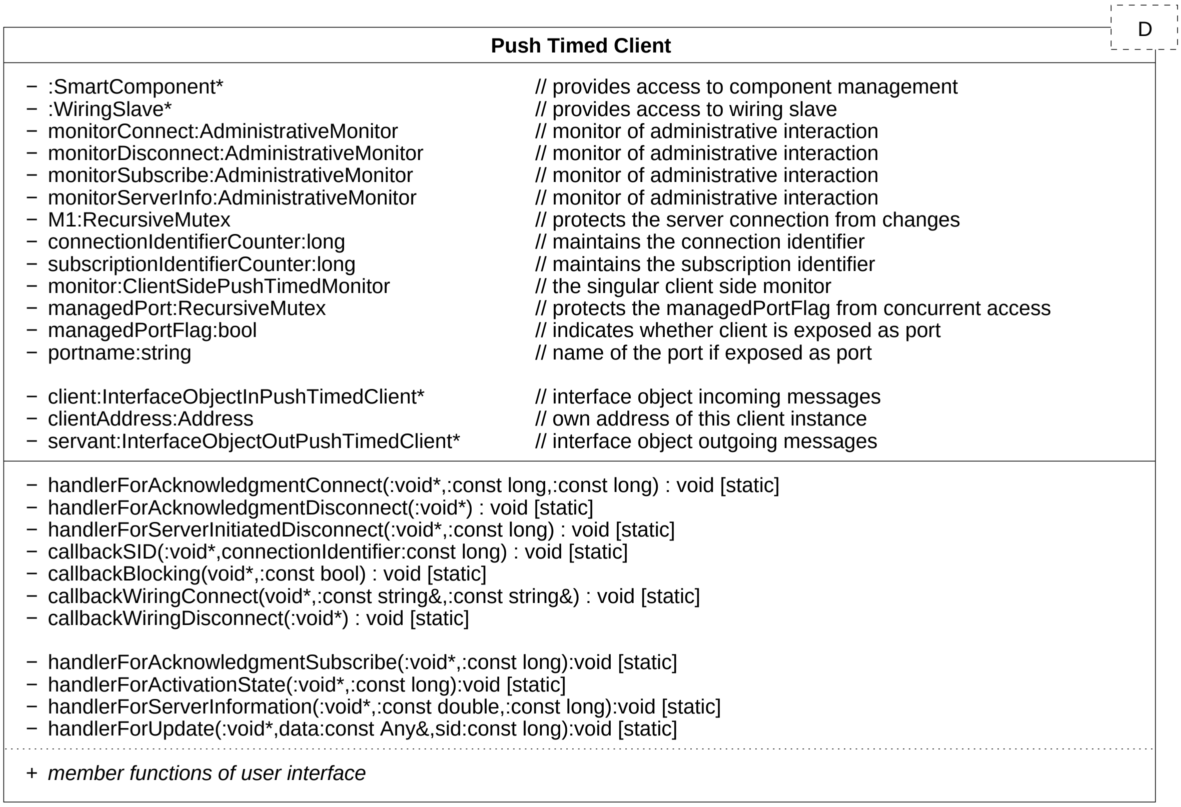
State after *wait* is passed	*cid*	*sid*	*update counter*	*disconnect flag*	Returned status	Situation
disconnected	x	x	x	x	*disconnected*	Client is currently disconnected so that the method invocation can get no update anymore.
connected	=	x	x	*false*	*unsubscribed*	Client got unsubscribed but is still connected and connection has not been modified. However, method invocation belongs to an outdated session.
	=	x	x	*true*	*disconnected*	Client gets disconnected and just performs the *unsubscribe*.
	<>	x	x	x	*disconnected*	Client got disconnected and thus also unsubscribed and then reconnected, perhaps even to a different service provider. Method invocation belongs to an outdated session.
subscribed	=	=	x	x	*cancelled*	The connection and the subscription have not been changed and there is still no update available so that blocking mode was set to *false* at least once while waiting so that one has to return.
	=	<>	x	x	*unsubscribed*	Client got unsubscribed and again subscribed so that the method invocation belongs to an outdated session.
	<>	x	x	x	*disconnected*	Client got disconnected and again connected and subscribed so that the method invocation belongs to an outdated session.
data	=	=	=	x	*cancelled*	No new update arrived and nothing changed with respect to the connection and the subscription so that blocking mode was set to *false* at least once while waiting.
	=	=	<>	x	*ok*	Received an update.
	=	<>	x	x	*unsubscribed*	Client got unsubscribed and subscribed again and even an update was already received but method invocation belongs to an outdated session.
	<>	x	x	x	*disconnected*	Client got disconnected, reconnected and subscribed again and even an update was already received but method invocation belongs to an outdated session.

Table 5.57: *The various situations that can be encountered after passing a* wait.

5.6.7.5 The Push Timed Pattern

The internals of the *push timed* pattern are almost identical to that of the *push newest* pattern. The major difference is the acknowledgment message for the *subscribe* that is needed to return the current activation state of the server. Furthermore, the server informs any subscribed client about getting activated or deactivated by means of an administrative (C/U) interaction and the client can get the current activation state and the cycle time of the server by means of an administrative (B/U) interaction.

The Client Part The client side internals of the *push timed client* class are shown in figure 5.147. The activation state and the cycle time is queried by means of an administrative (B/U) interaction so that this interaction cannot be interrupted. That does not matter since the server part solely reads the server state and the cycle time. Using an administrative (B/U) interaction has the advantage that at a client always only one *getServerInfo* method is active at a time which saves from inadequate overhead. The *getServerInfo* does not interact with the monitor of the service related interaction and solely returns the answer to the user level.



```
D
                                   Push Timed Client

−  :SmartComponent*                              // provides access to component management
−  :WiringSlave*                                 // provides access to wiring slave
−  monitorConnect:AdministrativeMonitor          // monitor of administrative interaction
−  monitorDisconnect:AdministrativeMonitor       // monitor of administrative interaction
−  monitorSubscribe:AdministrativeMonitor        // monitor of administrative interaction
−  monitorServerInfo:AdministrativeMonitor       // monitor of administrative interaction
−  M1:RecursiveMutex                             // protects the server connection from changes
−  connectionIdentifierCounter:long              // maintains the connection identifier
−  subscriptionIdentifierCounter:long            // maintains the subscription identifier
−  monitor:ClientSidePushTimedMonitor            // the singular client side monitor
−  managedPort:RecursiveMutex                    // protects the managedPortFlag from concurrent access
−  managedPortFlag:bool                          // indicates whether client is exposed as port
−  portname:string                               // name of the port if exposed as port

−  client:InterfaceObjectInPushTimedClient*      // interface object incoming messages
−  clientAddress:Address                         // own address of this client instance
−  servant:InterfaceObjectOutPushTimedClient*    // interface object outgoing messages

−  handlerForAcknowledgmentConnect(:void*,:const long,:const long) : void [static]
−  handlerForAcknowledgmentDisconnect(:void*) : void [static]
−  handlerForServerInitiatedDisconnect(:void*,:const long) : void [static]
−  callbackSID(:void*,connectionIdentifier:const long) : void [static]
−  callbackBlocking(void*,:const bool) : void [static]
−  callbackWiringConnect(void*,:const string&,:const string&) : void [static]
−  callbackWiringDisconnect(:void*) : void [static]

−  handlerForAcknowledgmentSubscribe(:void*,:const long):void [static]
−  handlerForActivationState(:void*,:const long):void [static]
−  handlerForServerInformation(:void*,:const double,:const long):void [static]
−  handlerForUpdate(:void*,data:const Any&,sid:const long):void [static]

+  member functions of user interface
```

***Figure 5.147:** Details of the internals of the* push timed client *class.*

Figure 5.148 shows the client side monitor of the *push timed* pattern. Again, the monitor exists as long as the client instance exists and its overall structure is the same as that of the *push newest* pattern. Both, the update counter and the activation counter are used by the *getUpdateWait* method. The *wait* method of the monitor blocks only if the state of the automaton is either *active* or *data* since these are the only states in which another update can be expected.

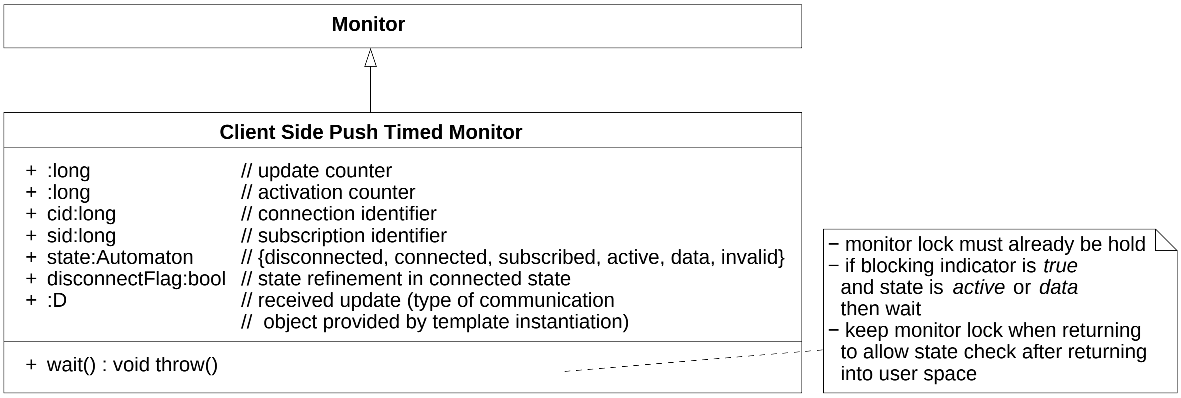

Monitor
Client Side Push Timed Monitor
+ :long // update counter
+ :long // activation counter
+ cid:long // connection identifier
+ sid:long // subscription identifier
+ state:Automaton // {disconnected, connected, subscribed, active, data, invalid}
+ disconnectFlag:bool // state refinement in connected state
+ :D // received update (type of communication
// object provided by template instantiation)
+ wait() : void throw()
− monitor lock must already be hold
− if blocking indicator is *true* and state is *active* or *data* then wait
− keep monitor lock when returning to allow state check after returning into user space

Figure 5.148: *The client side monitor of the* push timed *pattern.*

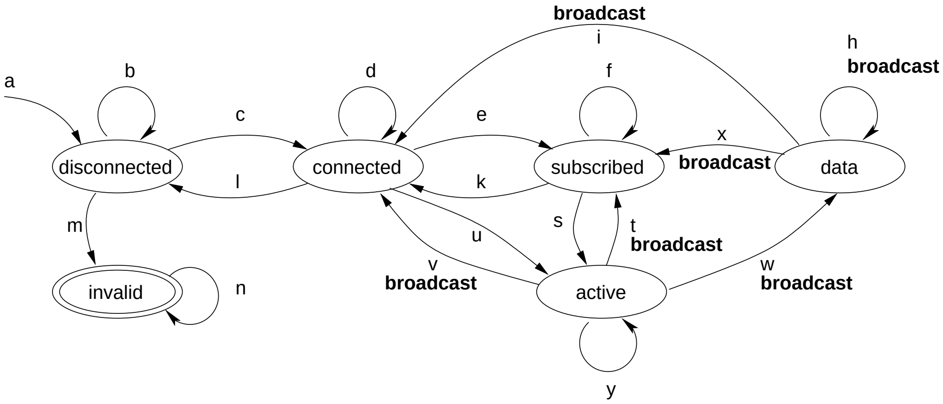


Figure 5.149: *The client side state automaton to coordinate the* push timed *client.*

The client side state automaton is shown in figure 5.149. Compared to the state automaton of the *push newest* pattern, it is extended by the *active* state and the *subscribed* state carries a different semantics. Since the client can get updates only as long as the server is active, one additionally has to distinguish a *passive* and an *active* server. In the *subscribed* state, the client is connected and subscribed to a passive server. In the *active* state, the client is connected and subscribed to an active server but there are no updates yet available. In the *data* state, the client is connected and subscribed to an active server and at least one update has been received yet.

The *subscribe* performs the transition *e* or *u* depending on the activation state of the server as returned by the acknowledgment of the *subscribe*. In case the server state switches from *active* to *passive*, the upcall handler for the *activation state* message performs the transition *t* and *x*, respectively. In case of switching from *passive* to *active*, the transition *s* is executed. An *unsubscribe* performs one of the transitions *i*, *k* and *v*. A *disconnect* always first executes an *unsubscribe*. The transitions *w* and *h* are performed when receiving an update.

The *data* state is the only state in which the *getUpdate* methods returns a valid update. The *getUpdateWait* method blocks in the *active* state to wait for the first update and in the *data* state to wait for the next update. Thus, the transitions *v*, *t*, *w* and *i* and *x* have to broadcast a signal. The transition *h* broadcasts a signal since the state automaton remains in the *data* state with receiving an update but the *getUpdateWait* method has to be notified. In case of the transitions *w* and *h*, the expected update arrived. In case of the transitions *i* and *v*, an *unsubscribe* occurred. Thus, from now on no more updates arrive that belong to the subscription phase in which the *getUpdateWait* method was invoked. In case of *t* and *x*, the server got into the *passive* mode and the regular update cycle is disturbed. In all cases, returning from the method call is the desired behavior. Thus, the client side state automaton of the *push timed* pattern and the *wait* method of the monitor comply with the general policy of the monitors.

The Client Side Administration Internals The *subscribe* message specifies the update rate in terms of whole-numbered multiples of the server update cycle and awaits the acknowledgment message to know whether to perform the state transition *e* or *u*. While waiting for the acknowledgment message, there can be no messages from the server that report the activation state and that would interfere with the state transitions of the *subscribe* procedure. The activation state is reported to subscribed clients only. Thus, a client gets a report on the activation state earliest after the upcall of the *subscribe* message at the server released the mutex M10. Then, however, the acknowledgment message was already sent and due to the kept order of messages, both cannot interfere.

The handler for the *activationState* message acquires the monitor lock to access the state automaton and then performs one of the transitions *s*, *t* or *x* as described above. Furthermore, it increments the activation counter that works in the same way as the update counter. It allows to detect that the server got into the *passive* mode in between which breaks the regular update rate.

The Client Side Service Internals Figure 5.150 shows the client side processing of the *update* message. If the update message contains a valid subscription identifier and if the state automaton is in the *active* or *data* state, then the received update is stored in the monitor and the state transition *w* or *h* is executed. It is important to note that no update message can be received in the *subscribed* state since that also indicates that the server is not active. The activation state of the server can be changed by the server. As soon as the server performs a state change, it reports that to all subscribed clients. Since messages keep their order and since the server does not send any updates in the *passive* state, there can be no late update messages after the *passive* state was reported. In principle, the check of the subscription identifier could also be omitted since in case of the *push timed* pattern, the *subscribe* possesses an acknowledgment. Thus, even in case of an *unsubscribe* and a *subscribe* in quick succession, there can be no outdated update message after the *subscribe* was acknowledged. This again holds due to the kept order of messages. Outdated update messages could arrive only while the *subscribe* is in progress but would then be rejected by the *connected* state. The *subscribe* procedure sets the state automaton to *subscribed* or *active* only after the acknowledgment arrived.

The *getUpdate* method does not perform any state transitions but checks the state to decide on the result of the method invocation. It returns *error* in case the automaton is in the *invalid* state, *disconnected* in the *disconnected* state, *unsubscribed* or *disconnected* in the *connected* state depending on the *disconnect flag*, *not activated* in the *subscribed* state and *no data* in the *active* state. It returns the available data with *ok* in the *data* state.

The *getUpdateWait* method has the same structure as the *getUpdateWait* method of the *push newest* pattern and its blocking behavior is summarized in table 5.58. Again, one has to carefully check the situation after having passed the *wait*. In contrast to the *push newest* pattern, one now also has to take into account the activation counter. Tables 5.59 and 5.60 summarize the various combinations and how they are handled.

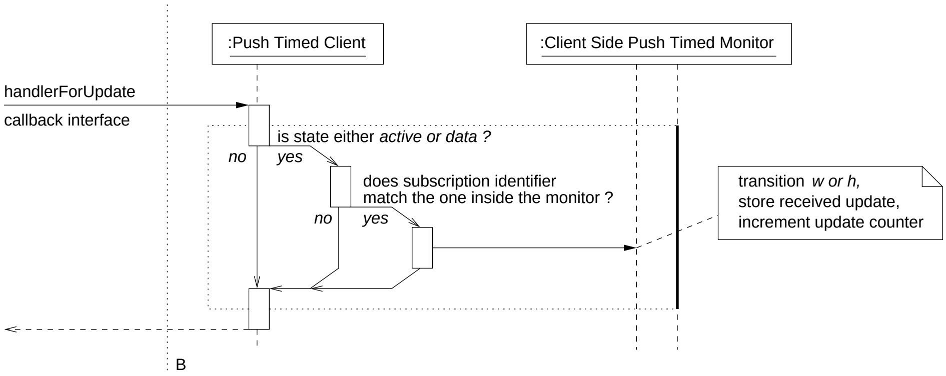


Figure 5.150: *The internals of the client side handler for the update message.*

Automaton state on invocation	Action
data	block in *wait* of monitor
active	block in *wait* of monitor
subscribed	rush through *wait*
connected	rush through *wait*
disconnected	rush through *wait*
invalid	rush through *wait*

Table 5.58: *The blocking behavior of the* getUpdateWait *method.*

The Server Part The server side internals of the *push timed server* class are shown in figure 5.151. Again, no advanced monitors are used and the *list of clients* is enriched in the same way as in case of the *push newest* server. In addition, it contains the *update rate* and the *update counter* that manage the individual update rates.

The Server Side Administration Internals The server parts of the *connect*, the *disconnect* and the *server initiated disconnect* procedures of the *push timed* server comply with the generic form of those administrative interactions. All parts that are related to the *list of monitors* are removed. The handler for the *unsubscribe* message is the same as the one of the *push newest* pattern. The *subscribe* message additionally provides the individual update rate for the subscription. The handler of the *subscribe* message initializes the update counter to 1 so that the client gets the next available update. The handler returns the current activation state by sending the acknowledgment message before it releases the mutex M10.

The handler to process the *getServerInformation* message acquires the mutex M10, reads the values of the activation state and the cycle time and then sends the response before it releases the mutex M10.

The Server Side Service Internals The *start* method registers the *push timed* server at the timer of the component management class and notifies all subscribed clients about the state change. The timer invokes the static *callback timer* method which then invokes the user provided handler. It depends on the load of the user provided handler whether one can use a passive handler since one

State after *wait* is passed	*cid*	*sid*	*activation counter*	*update counter*	*disconnect flag*	Returned status	Situation
disconnected	x	x	x	x	x	*disconnected*	Client is currently disconnected so that the method invocation cannot get any update anymore.
connected	=	x	x	x	*false*	*unsubscribed*	Client got unsubscribed but is still connected and connection has not been modified. However, method invocation belongs to an outdated session.
	=	x	x	x	*true*	*disconnected*	Client gets disconnected and just performs the *unsubscribe*.
	<>	x	x	x	x	*disconnected*	Client got disconnected and thus also unsubscribed and then reconnected, perhaps even to a different service provider. Method invocation belongs to an outdated session.
subscribed	=	=	=	x	x	—	(not possible since the *subscribed* state can be reached only with at least a different *cid*, *sid* or *activation counter*).
	=	=	<>	x	x	*not activated*	The connection and the subscription have not been changed but the server got deactivated meanwhile so that the update cycle is noncontinuous and one has to return.
	=	<>	x	x	x	*unsubscribed*	Client got unsubscribed and again subscribed so that the method invocation belongs to an outdated session.
	<>	x	x	x	x	*disconnected*	Client got disconnected and again connected and subscribed so that the method invocation belongs to an outdated session.

Table 5.59: *The various situations that can be encountered after passing a* wait *- part one.*

must not block the timer activities. The *start* method always first invokes a *stop*.

The handler invocation is solely a notification that another update is due and only the *put* method sends updates to subscribed clients. The proper update rate depends on properly calling the *put* method. In case the server is activated, it iterates through the list of subscribed clients and decrements the individual update counters. An update is sent to a subscribed client if its update counter reaches zero. The update counter is then reinitialized to the client provided update rate.

In case of an active server, the *stop* method removes the timer callback from the component's timer and notifies the subscribed clients on the state change. Furthermore, it resets all update counters to 1 so that subscribed clients get the first update in case of reactivation.

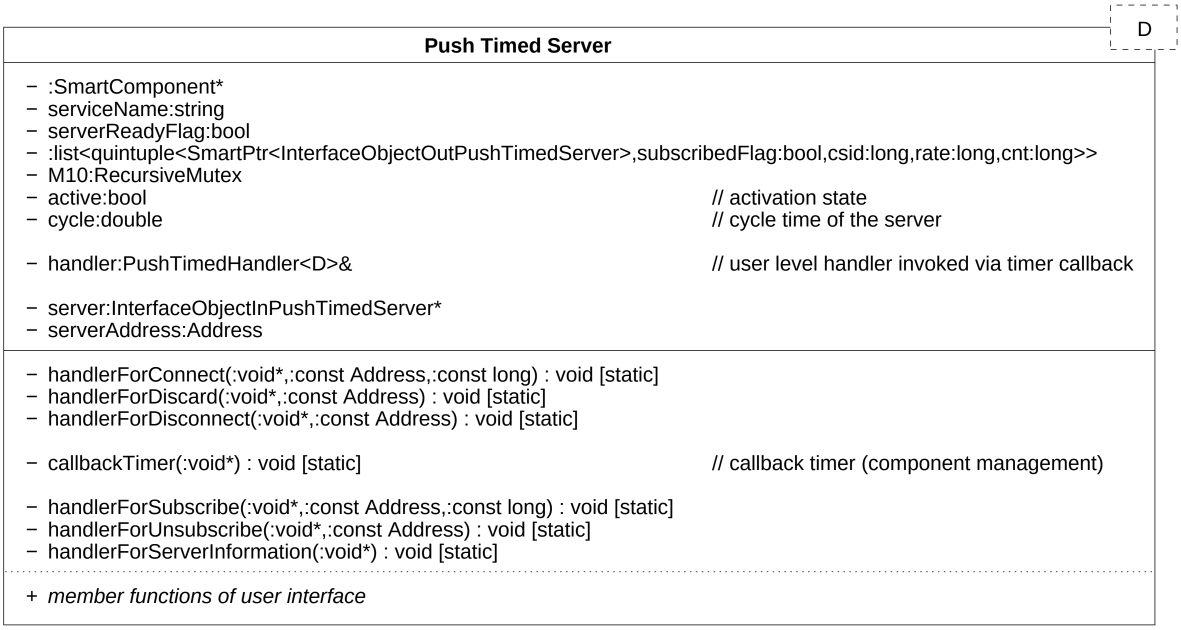

D
Push Timed Server
− :SmartComponent*
− serviceName:string
− serverReadyFlag:bool
− :list<quintuple<SmartPtr<InterfaceObjectOutPushTimedServer>,subscribedFlag:bool,csid:long,rate:long,cnt:long>>
− M10:RecursiveMutex
− active:bool // activation state
− cycle:double // cycle time of the server
− handler:PushTimedHandler<D>& // user level handler invoked via timer callback
− server:InterfaceObjectInPushTimedServer*
− serverAddress:Address
− handlerForConnect(:void*,:const Address,:const long) : void [static]
− handlerForDiscard(:void*,:const Address) : void [static]
− handlerForDisconnect(:void*,:const Address) : void [static]
− callbackTimer(:void*) : void [static] // callback timer (component management)
− handlerForSubscribe(:void*,:const Address,:const long) : void [static]
− handlerForUnsubscribe(:void*,:const Address) : void [static]
− handlerForServerInformation(:void*) : void [static]
+ member functions of user interface

Figure 5.151: *Details of the internals of the* push timed server *class.*

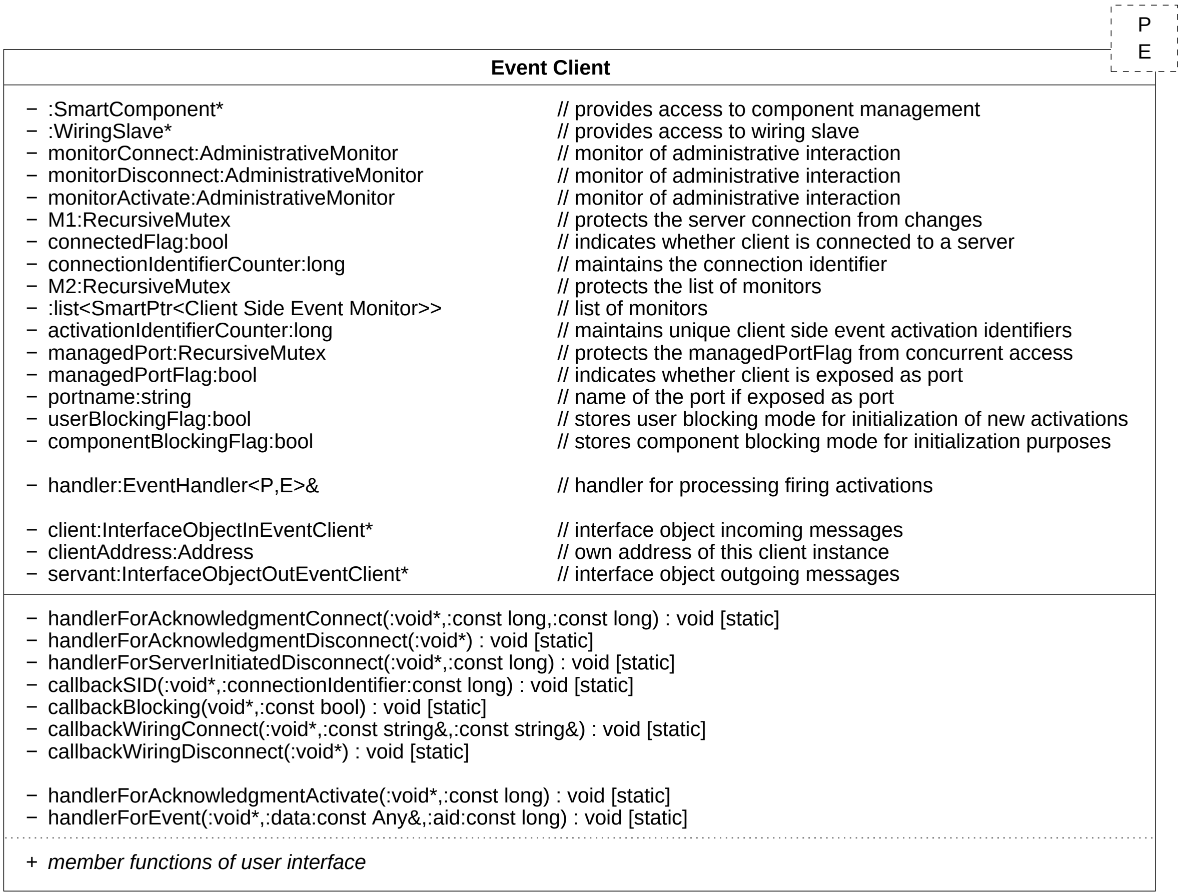
State after *wait* is passed	*cid*	*sid*	*activation counter*	*update counter*	*disconnect flag*	Returned status	Situation
active	=	=	=	x	x	*cancelled*	The connection, the subscription and the activation have not been changed so that the blocking mode was set to *false* at least once while waiting for the first update from the server so that one has to return.
	=	=	<>	x	x	*not activated*	Server got deactivated and activated again so that the update cycle is noncontinuous and one has to return.
	=	<>	x	x	x	*unsubscribed*	Client got unsubscribed and subscribed again and no update is yet received but method invocation belongs to an outdated session.
	<>	x	x	x	x	*disconnected*	Client got disconnected, reconnected and subscribed again and server is active but method invocation belongs to an outdated session.
data	=	=	=	=	x	*cancelled*	No new update arrived and nothing changed with respect to the connection, the subscription and the activation so that blocking mode was set to *false* at least once while blocking on the *wait*.
	=	=	=	<>	x	*ok*	Received an update.
	=	=	<>	x	x	*not activated*	The connection and the subscription have not been changed but the server got deactivated meanwhile so that the update cycle is noncontinuous and one has to return.
	=	<>	x	x	x	*unsubscribed*	Client got unsubscribed and subscribed again and even an update was already received but method invocation belongs to an outdated session.
	<>	x	x	x	x	*disconnected*	Client got disconnected, reconnected and subscribed again and even an update was already received but method invocation belongs to an outdated session.

Table 5.60: *The various situations that can be encountered after passing a* wait *- part two.*

5.6.7.6 The Event Pattern

The internals of the *event* pattern are best understood by comparing its client part with the client part of the *query* pattern and its server part with the server part of a *push* pattern. At the client side, each event activation has its own monitor instance since an event can handle any number of concurrent activations each with its own parameters. At the server side, the *put* method triggers the evaluation of the event condition which decides for each activation whether to fire. However, in contrast to the *push* server, it contains the full server side structures to handle an arbitrary number of activations per client. These are comparable to the structures of the *query* server pattern.

The Client Part The client side internals of the *event client* class are shown in figure 5.152. Since a client can invoke any number of event activations, the server has to be able to reject activations. Thus, the activation is based on an administrative (B/U) interaction that returns a status indicating whether the activation was accepted. Each event activation has its own monitor dynamically generated monitor instance. These are all listed in the *list of monitors*. Each activation has a unique activation identifier that compares to the unique identifiers of the active queries of the *query* pattern.



```
Event Client                                                                    P
                                                                                E

− :SmartComponent*                                 // provides access to component management
− :WiringSlave*                                    // provides access to wiring slave
− monitorConnect:AdministrativeMonitor             // monitor of administrative interaction
− monitorDisconnect:AdministrativeMonitor          // monitor of administrative interaction
− monitorActivate:AdministrativeMonitor            // monitor of administrative interaction
− M1:RecursiveMutex                                // protects the server connection from changes
− connectedFlag:bool                               // indicates whether client is connected to a server
− connectionIdentifierCounter:long                 // maintains the connection identifier
− M2:RecursiveMutex                                // protects the list of monitors
− :list<SmartPtr<Client Side Event Monitor>>       // list of monitors
− activationIdentifierCounter:long                 // maintains unique client side event activation identifiers
− managedPort:RecursiveMutex                       // protects the managedPortFlag from concurrent access
− managedPortFlag:bool                             // indicates whether client is exposed as port
− portname:string                                  // name of the port if exposed as port
− userBlockingFlag:bool                            // stores user blocking mode for initialization of new activations
− componentBlockingFlag:bool                       // stores component blocking mode for initialization purposes

− handler:EventHandler<P,E>&                       // handler for processing firing activations

− client:InterfaceObjectInEventClient*             // interface object incoming messages
− clientAddress:Address                            // own address of this client instance
− servant:InterfaceObjectOutEventClient*           // interface object outgoing messages

− handlerForAcknowledgmentConnect(:void*,:const long,:const long) : void [static]
− handlerForAcknowledgmentDisconnect(:void*) : void [static]
− handlerForServerInitiatedDisconnect(:void*,:const long) : void [static]
− callbackSID(:void*,:connectionIdentifier:const long) : void [static]
− callbackBlocking(void*,:const bool) : void [static]
− callbackWiringConnect(:void*,:const string&,:const string&) : void [static]
− callbackWiringDisconnect(:void*) : void [static]

− handlerForAcknowledgmentActivate(:void*,:const long) : void [static]
− handlerForEvent(:void*,:data:const Any&,:aid:const long) : void [static]

+ member functions of user interface
```

Figure 5.152: *Details of the internals of the* event *client class.*

Figure 5.153 shows the client side monitor of the *event* pattern. The *event mode* denotes the activation mode that is either *single* or *continuous*. The *firing counter* is used in *continuous* mode

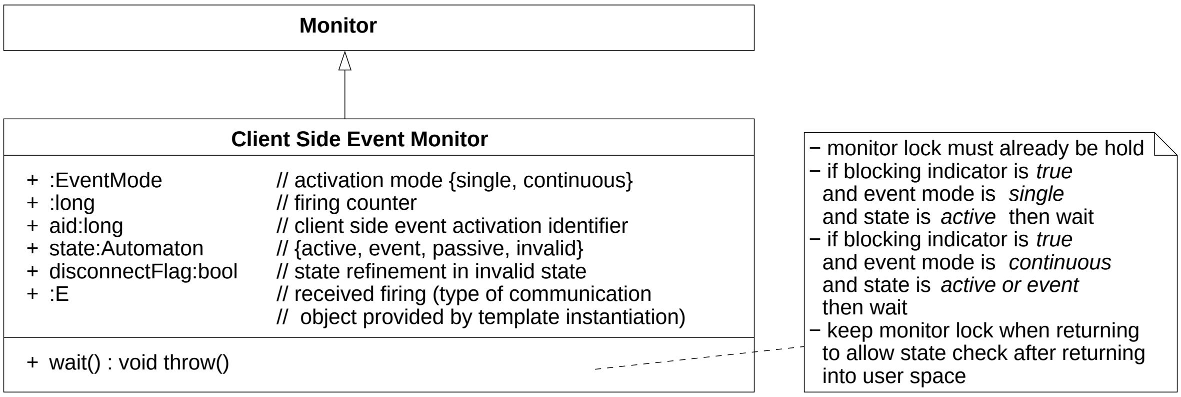

Monitor
Client Side Event Monitor
+ :EventMode // activation mode {single, continuous}
+ :long // firing counter
+ aid:long // client side event activation identifier
+ state:Automaton // {active, event, passive, invalid}
+ disconnectFlag:bool // state refinement in invalid state
+ :E // received firing (type of communication
// object provided by template instantiation)
+ wait() : void throw()
− monitor lock must already be hold
− if blocking indicator is *true* and event mode is *single* and state is *active* then wait
− if blocking indicator is *true* and event mode is *continuous* and state is *active or event* then wait
− keep monitor lock when returning to allow state check after returning into user space

Figure 5.153: *The client side representation of an event activation.*

in the same way as the *update counter* of the *push* patterns. The *disconnect flag* provides additional information in case of the *invalid* state to be able to distinguish between a *deactivate* and a *disconnect*. The *wait* method distinguishes between *single* and *continuous* mode since only in *continuous* mode more than one firing can occur.

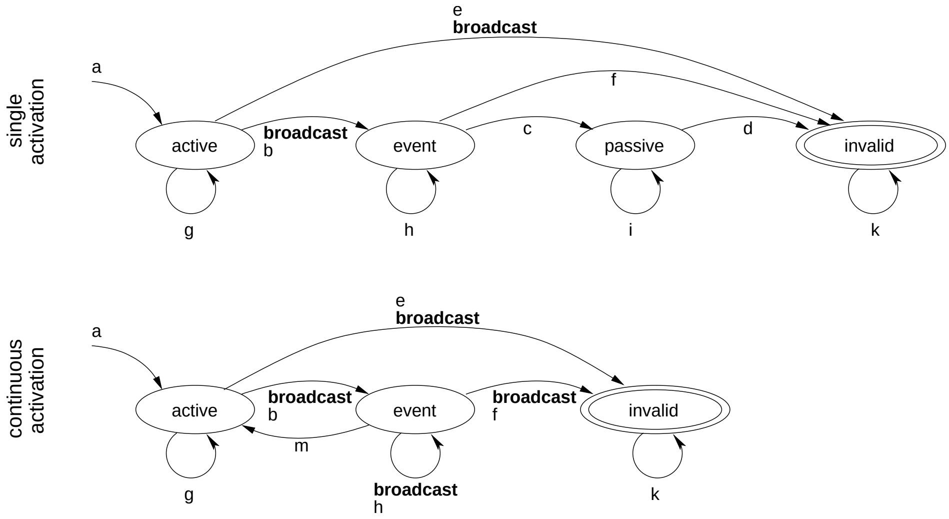


Figure 5.154: *The client side state automaton of an event activation.*

The client side state automaton is shown in figure 5.154. The *active* state indicates that no unconsumed firing is available and that a firing can occur. In the *event* state, an unconsumed firing is available. The *passive* state indicates that there is no unconsumed firing available and that there will be no more firing but the event is still activated. Finally, the *invalid* state indicates that the activation identifier is already invalid.

Lets first consider the state automaton of an activation in *single* mode. The *activate* method generates a new monitor instance that is initialized to the *active* state. The transition *b* is performed in case the event fires and the transition *c* in case the firing is consumed. The transitions *d*, *e* and *f* are performed by a *deactivate*. In principle, an activation identifier gets invalid with calling the *deactivate* method for it. All activations get invalid by a *disconnect*.

A firing can solely occur in the *active* state and the *wait* method of the monitor thus only blocks in the *active* state. Since the state transitions *b* and *e* lead from blocking to non-blocking states, these have to broadcast a signal. In case of the transition *b*, the firing arrived and in case of the transition *e*, a *deactivate* occurred.

Lets now consider the state automaton of an activation in *continuous* mode. The transition *b* is executed with the first firing and the transition *m* when a firing is consumed. The next firing then again executes the transition *b*. In case a firing has not yet been consumed when the next one arrives, it gets overwritten as indicated by the transition *h*. The transitions *e* and *f* are executed by a *deactivate*.

A firing can occur either in the *active* or the *event* state and thus, in *continuous* mode, the *wait* method of the monitor blocks in both states. Therefore, one needs to broadcast a signal with the state transitions *b*, *e* and *f* since these lead from blocking to non-blocking states and one can individually decide for the transitions *g*, *h* and *m* since these lead from blocking to blocking states. Due to the desired behavior of the blocking methods, solely the state transition *h* broadcasts a signal.

The Client Side Administration Internals The *add*, *remove*, *connect* and *disconnect* methods work like their equivalents at the other patterns. Since all event activations get invalid with a *disconnect*, the *disconnect* method removes all monitors from the list of monitors, sets their *disconnectFlag* to *true* and performs the state transition into the *invalid* state. Due to the *smart pointers*, the blocking methods can still access the monitor instance as long as required. The *disconnectFlag* allows to determine whether a *disconnect* or a *deactivate* is the reason for awakening in the *invalid* state. At the server part, the affected activations are identified by the client address in the same way as in case of the *query* pattern. This procedure is much faster than deactivating one activation after the other.

The *activate* method is shown in figure 5.155 and it compares to the *request* method of the *query* pattern. Each activation has its own dynamically generated monitor instance which is accessed by means of an activation identifier. The *deactivate* method shown in figure 5.156 compares to the *discard* method.

The Client Side Service Internals (Single Mode) The *try* method solely checks the state automaton and returns the appropriate status code. The *get* method performs the state transition *c* and consumes the firing when invoked in the *event* state and otherwise returns without a valid firing. The status code depends on the state of the state automaton.

The *getWait* and the *getNext* method get blocked only when invoked in the *active* state since the event fires only once. Thus, they get resumed either due to the signal of the blocking mode or due to the broadcast of *b* or *e*. Since *b* indicates the arrival of the expected firing and since *e* indicates a *deactivate*, in all cases the *wait* must not be reinvoked which complies with the policy of the monitors. The structure of the *getWait* and the *getNext* methods in the *single* mode is shown in figures 5.157 and 5.158.

In case the *wait* of the monitor awakes in the *active* state, the blocking method was cancelled and there is still no firing available. In the *event* state, an unconsumed firing is available and in case of the *passive* state, a concurrent method call with the same activation identifier already consumed the firing. In the *invalid* state, the activation identifier is not valid anymore either due to a *deactivate* or due to a *disconnect*.

The Client Side Service Internals (Continuous Mode) The *try* and the *get* method work in the same way for the continuous mode taking into account the differences of the state automatons. The *get* method, for example, now performs the state transition *m* in case it gets invoked in the *event* state.

In case the *getWait* method is invoked in the *event* state, it returns the unconsumed firing and in case of the *invalid* state, a *wrong identifier* status. In both cases, the *wait* method of the monitor is not invoked as shown in figure 5.159. The *getWait* method blocks only in the *active* state. Thus, it gets resumed either due to the signal of the blocking mode or due to the broadcast of *b* or *e*. Since *b* indicates the arrival of a firing and since *e* indicates a *deactivate*, in all cases the *wait* must not be reinvoked.

In case the *getNext* method is invoked in the *invalid* state, it also returns a *wrong identifier* status. However, it invokes the *wait* method in both the *active* and the *event* state as shown in figure 5.160. Thus, it gets resumed either by *h* or *f* or by a signal of the blocking mode. The transition *h* indicates that the next firing arrived and *f* notifies the *getNext* method about a *deactivate*. In all cases, the *wait* must not be reinvoked.

Both the *getWait* and the *getNext* method use the *firing counter* to discriminate various states after the *wait* of the monitor was passed. The different states are summarized in table 5.61. The table applies to both the *getWait* and the *getNext* method.

State after *wait* is passed	firing counter *fc*	Returned status	Situation
active	= saved *fc*	cancelled	No firing arrived so that blocking mode must have been set to *false* at least once while waiting.
	= saved *fc* + 1	lost	A firing arrived but a concurrent call with the same activation identifier consumed the firing which is thus lost.
	> saved *fc* + 1	missed/ok	Several firings arrived and got consumed so that the expected one was missed. This can happen due to the Mesa-style semantics of the condition variable. In case the *getWait* and *getNext* methods shall return the very next event firing, this indicates a missed event firing, otherwise one simply takes the most recent event firing.
event	= saved *fc*	cancelled	No firing arrived so that blocking mode must have been set to *false* at least once while waiting. Applies only to the *getNext* method.
	= saved *fc* +1	ok	The expected firing arrived.
	> saved *fc* + 1	missed/ok	The very next firing is already overwritten by a successive one so that one missed an event firing (see above).
invalid	disconnectFlag *true*	disconnected	Activation got deactivated due to a *disconnect*, thus return *disconnected* instead of *not activated*.
	disconnectFlag *false*	not activated	Activation got deactivated.

Table 5.61: *The various situations that can be encountered by the* getWait *and the* getNext *methods in the* continuous *mode after passing a* wait.

Now, it is also obvious why one is not allowed to broadcast a signal with the state transition *m*. If the *getNext* method is invoked on the *event* state, consuming the available firing with the *get* or *getWait* method would wrongly unblock the waiting *getNext* method. Of course, it is never a good idea to use the same activation identifier in concurrent calls.

Remark The client side callback handler for incoming firings corresponds to the callback handler for answers of the *query* pattern. In case a handler is provided with the instantiation of the *event* client pattern, the handler is invoked with every firing of each activation. The different activations can easily be sorted out by means of the activation identifier. Since that handler is located at the client part, the strict rules for client side handlers apply.

The Server Part The server side internals of the *event server* class are shown in figure 5.161. The first argument of the handler for the *activate* message is the *this*-pointer, the second argument the address of the client that performs the activate, the third argument is the event mode followed by the client side activation identifier and the last argument is the content of the communication object that provides the parameters for the event predicate. The third argument of the handler for the *deactivate* message is the client side activation identifier that is needed to deactivate the proper activation.

All activations are represented by their own monitor instance that is shown in figure 5.162. At the *event* server, there are currently no blocking methods so that the monitor is again solely used to coordinate the access to its attributes.

Figure 5.163 shows the server side state automaton of an event activation. In *single* mode, the transition *o* is performed with the first firing. In the *invalid* state, it is not considered by subsequent tests of the event predicate. In *continuous* mode, the transition *n* is performed with each firing. Both a *deactivate* and a *disconnect* always result in the *invalid* state.

The Server Side Administration Internals The server parts of the *connect*, the *disconnect* and the *server initiated disconnect* of the *event* server comply with the generic form of those administrative interactions. The handler for the *disconnect* iterates through the *list of monitors* and sets the state automatons of all entries belonging to that client to *invalid* before it removes the monitor instance from the list. The *deactivate* does the same but only for the monitor instance with the matching client address and activation identifier. The *activate* generates a new monitor instance, initializes it and adds it to the *list of monitors* so that the new activation is considered by the next test of the event predicate. The *activate* handler also sends the acknowledgment message.

The Server Side Service Internals At the server side, the *put* method provides the current state that is to be tested against the activation parameters. The *put* method acquires the mutex M10 and iterates through all activations. For each activation, it invokes the user provided handler that implements the event predicate as already illustrated in figure 5.46. Each activation is tested with its individual activation parameters. Only in case the handler returns *true*, the event fires and sends the *event* message with the event communication object E that carries individual results.

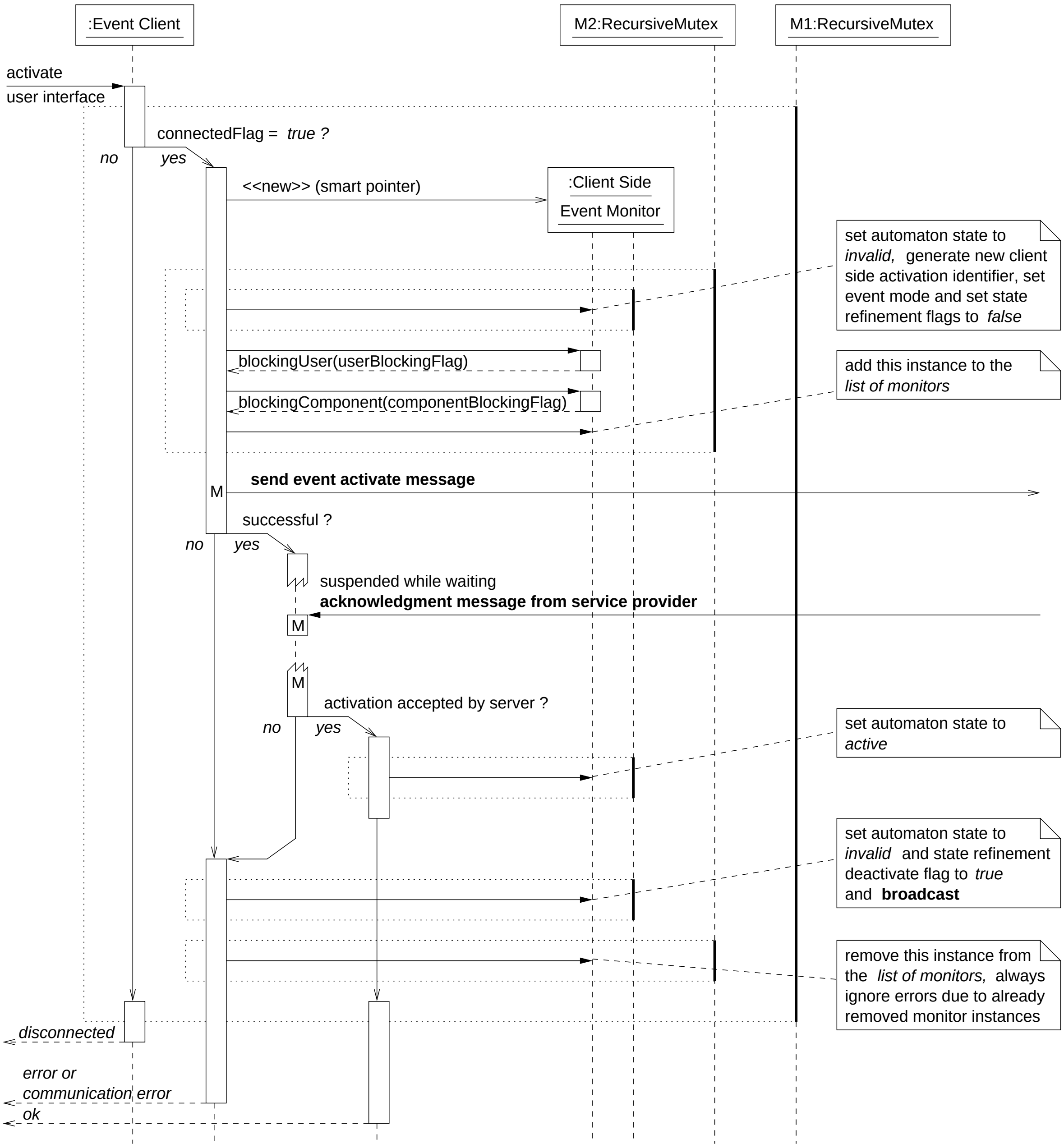

:Event Client
M2:RecursiveMutex
M1:RecursiveMutex
activate
user interface
connectedFlag = true ?
no
yes
<<new>> (smart pointer)
:Client Side
Event Monitor
set automaton state to invalid, generate new client side activation identifier, set event mode and set state refinement flags to false
blockingUser(userBlockingFlag)
blockingComponent(componentBlockingFlag)
add this instance to the list of monitors
M
send event activate message
successful ?
no
yes
suspended while waiting
acknowledgment message from service provider
M
M
activation accepted by server ?
no
yes
set automaton state to active
set automaton state to invalid and state refinement deactivate flag to true and broadcast
remove this instance from the list of monitors, always ignore errors due to already removed monitor instances
disconnected
error or
communication error
ok

Figure 5.155: *The internals of the client side* activate *method.*

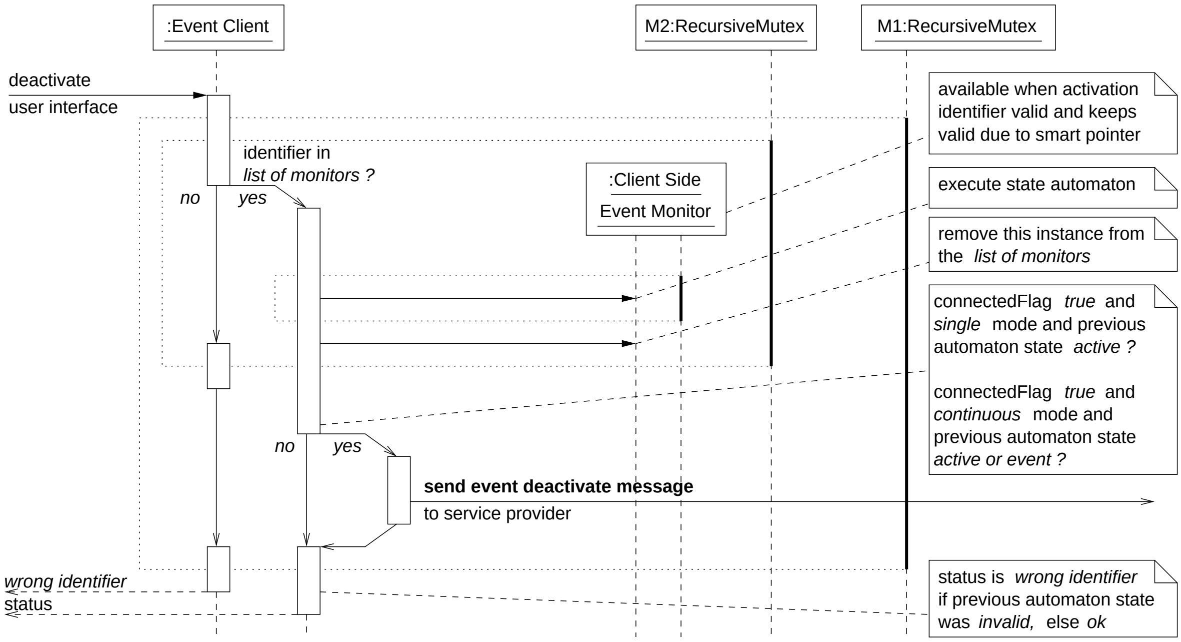

:Event Client
M2:RecursiveMutex
M1:RecursiveMutex
deactivate
user interface
available when activation identifier valid and keeps valid due to smart pointer
identifier in
list of monitors ?
no
yes
:Client Side
Event Monitor
execute state automaton
remove this instance from the *list of monitors*
connectedFlag *true* and *single* mode and previous automaton state *active ?*
connectedFlag *true* and *continuous* mode and previous automaton state *active or event ?*
no
yes
send event deactivate message
to service provider
wrong identifier
status
status is *wrong identifier* if previous automaton state was *invalid,* else *ok*

Figure 5.156: *The internals of the client side* deactivate *method.*

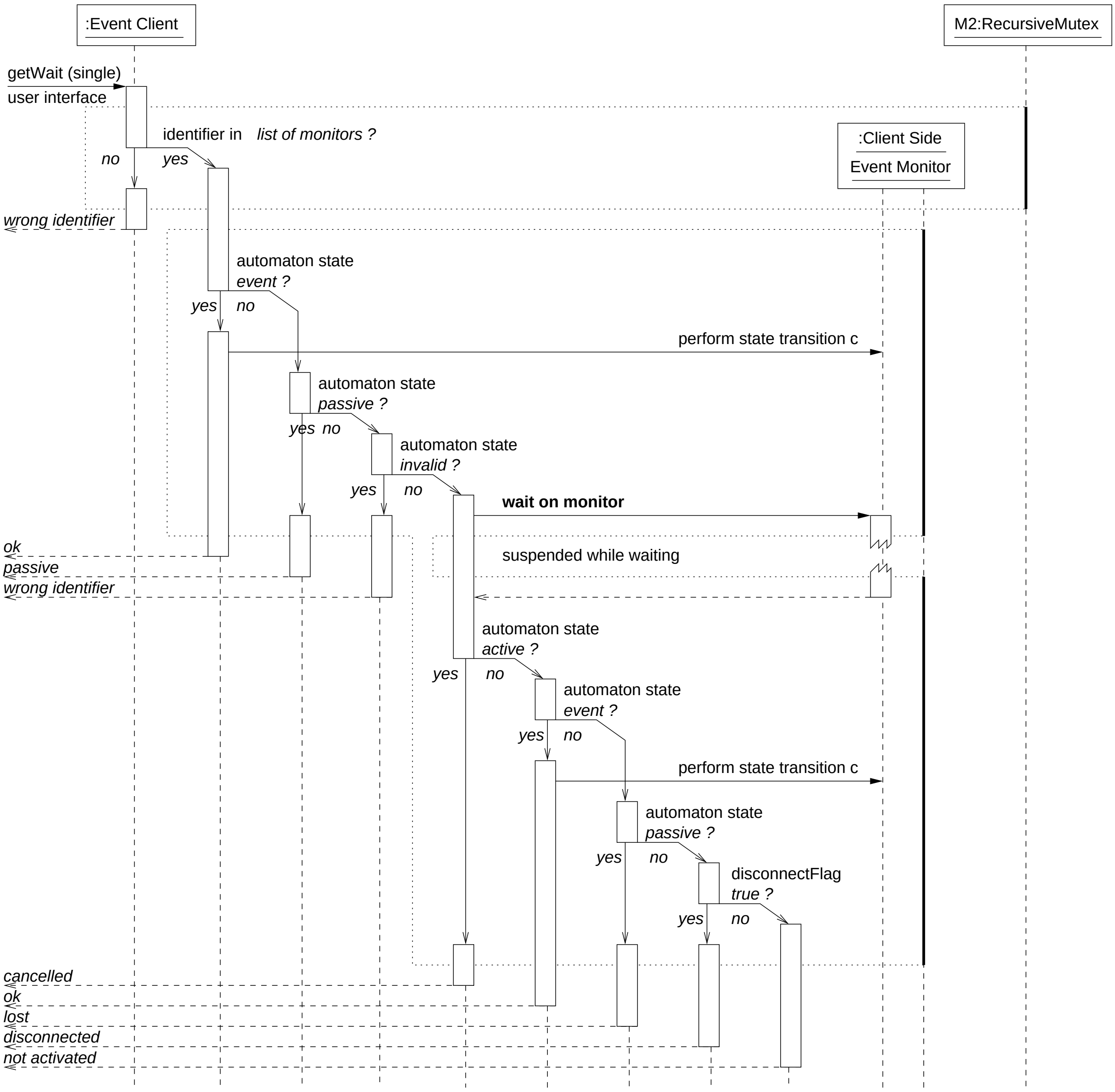


Figure 5.157: *The internals of the client side* getWait *method in* single *mode.*

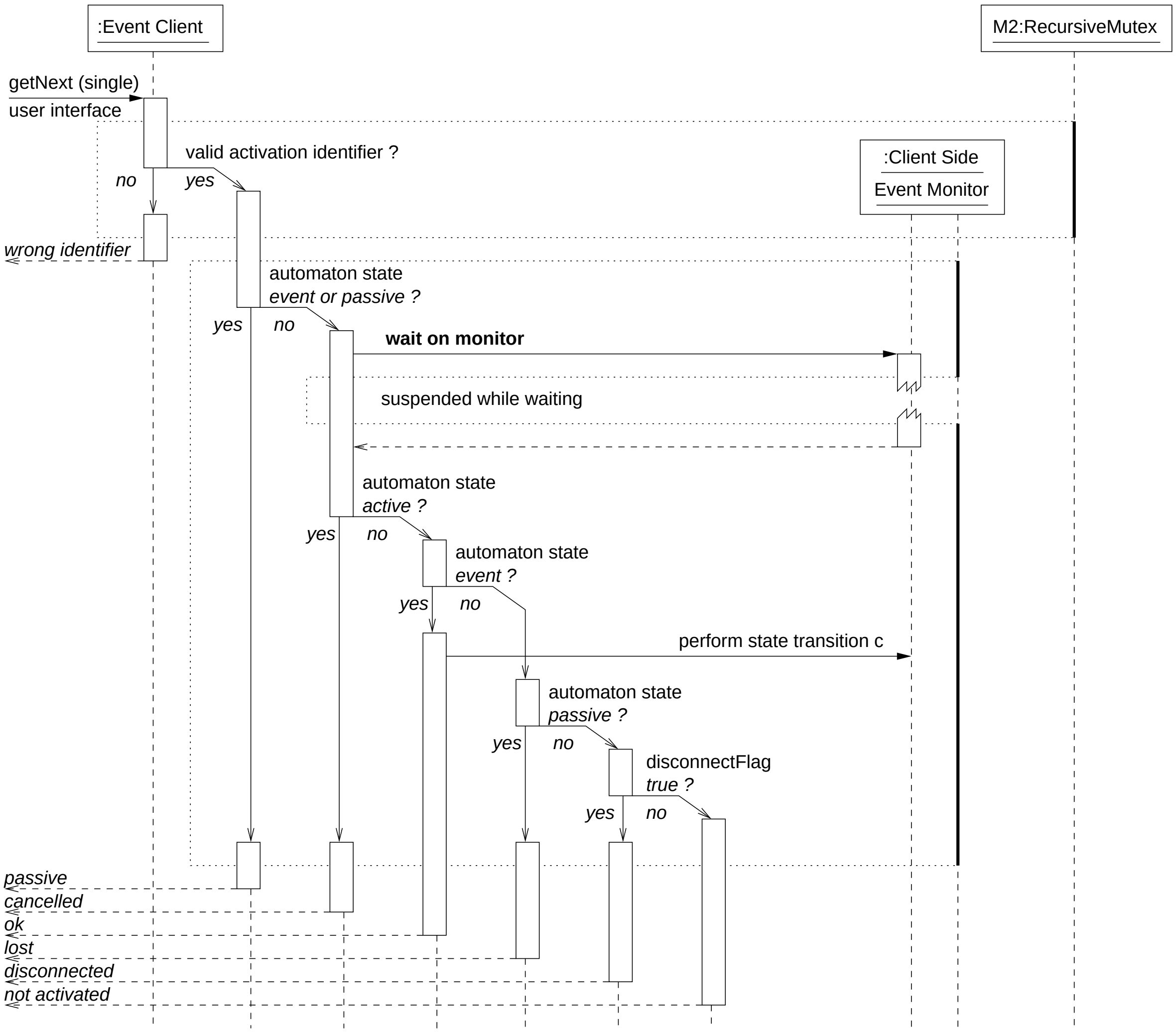


Figure 5.158: *The internals of the client side* getNext *method in* single *mode.*

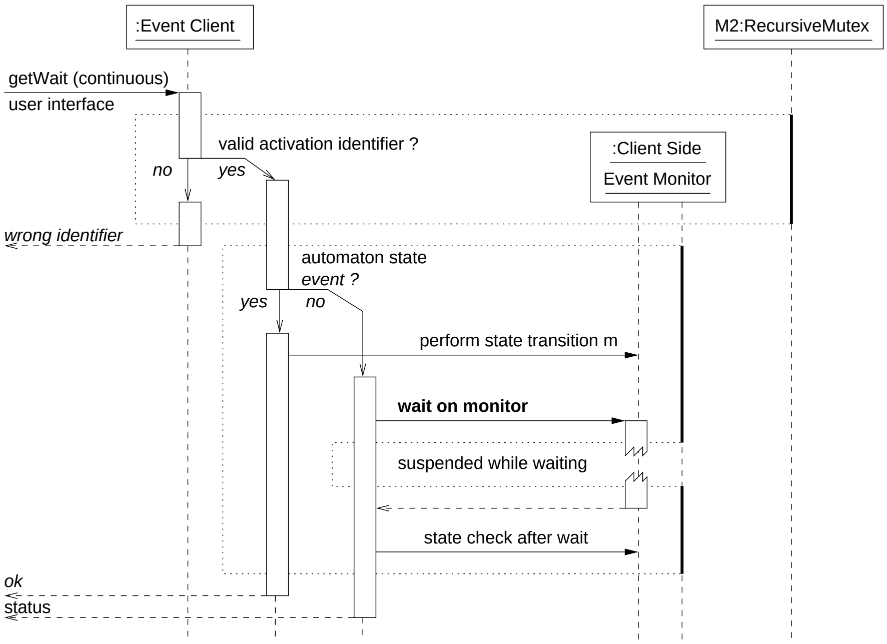


Figure 5.159: The internals of the client side getWait *method in* continuous *mode.*

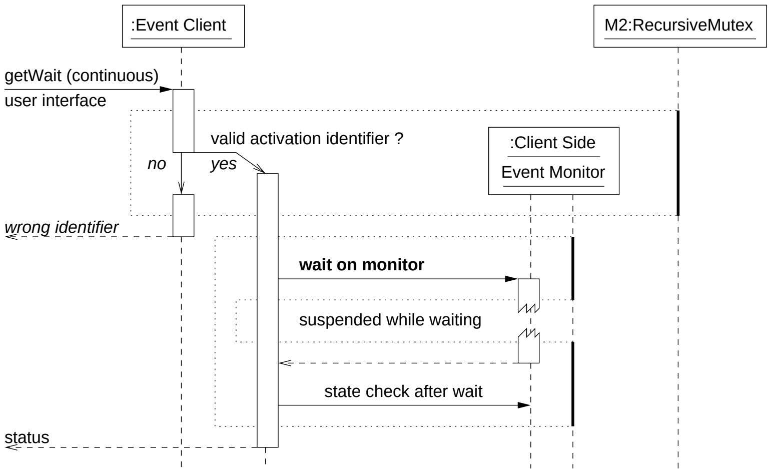


Figure 5.160: The internals of the client side getNext *method in* continuous *mode.*

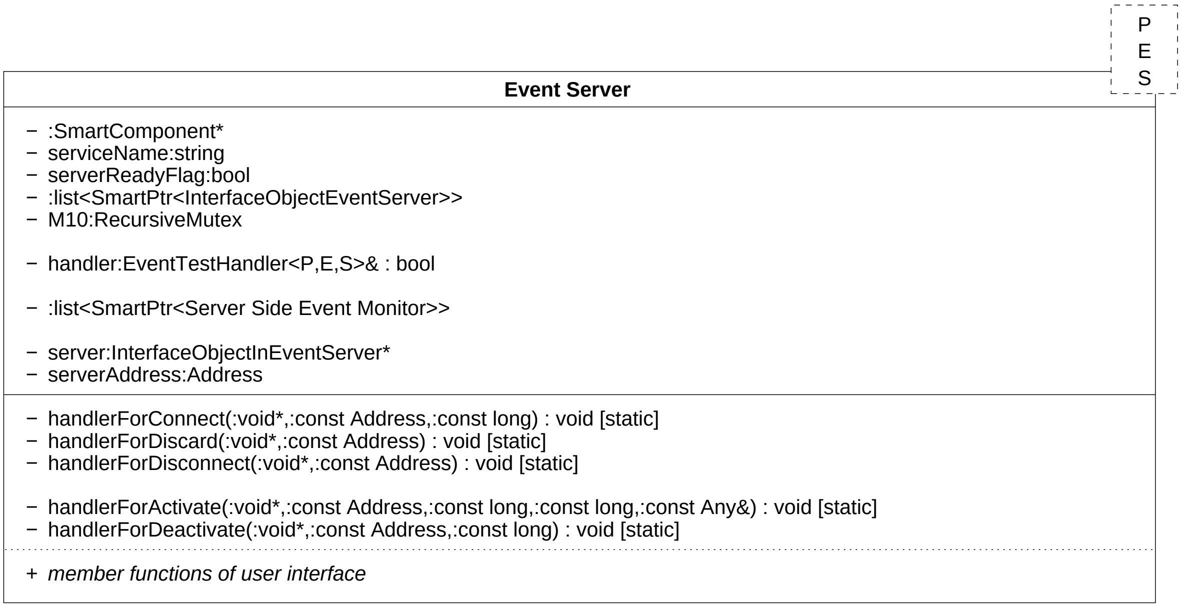

P
E
S
Event Server
− :SmartComponent*
− serviceName:string
− serverReadyFlag:bool
− :list<SmartPtr<InterfaceObjectEventServer>>
− M10:RecursiveMutex
− handler:EventTestHandler<P,E,S>& : bool
− :list<SmartPtr<Server Side Event Monitor>>
− server:InterfaceObjectInEventServer*
− serverAddress:Address
− handlerForConnect(:void*,:const Address,:const long) : void [static]
− handlerForDiscard(:void*,:const Address) : void [static]
− handlerForDisconnect(:void*,:const Address) : void [static]
− handlerForActivate(:void*,:const Address,:const long,:const long,:const Any&) : void [static]
− handlerForDeactivate(:void*,:const Address,:const long) : void [static]
+ *member functions of user interface*

Figure 5.161: *Details of the internals of the* event *server class.*

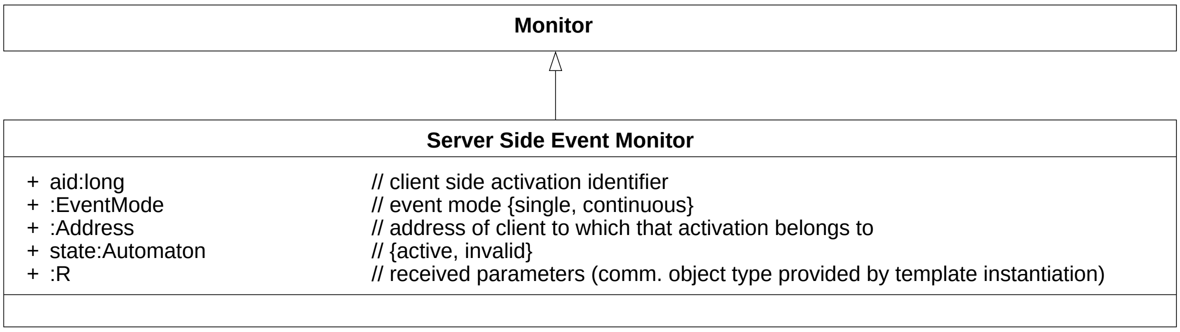

Monitor
Server Side Event Monitor
+ aid:long // client side activation identifier
+ :EventMode // event mode {single, continuous}
+ :Address // address of client to which that activation belongs to
+ state:Automaton // {active, invalid}
+ :R // received parameters (comm. object type provided by template instantiation)

Figure 5.162: *The server side representation of an event activation.*

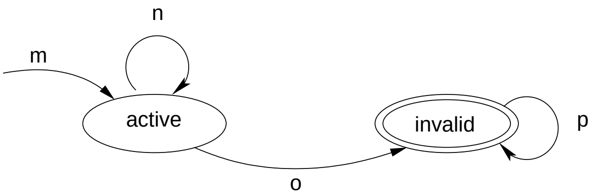


Figure 5.163: *The server side state automaton to manage the lifecycle of an event activation.*

5.6.7.7 The Wiring Pattern

The *wiring* pattern is different to the other communication patterns in not requiring communication objects. Each component can possess at most one *wiring slave* that handles all ports of a component. In contrast to the other communication patterns, a *wiring master* needs not to be connected to a *wiring slave* before it can access the ports managed by it. In fact, the connection to the *wiring slave* that holds the denoted port is established and removed on-the-fly and seamless to the user. The wiring pattern internally applies the *query* pattern to implement the interaction of the wiring master and a wiring slave. The wiring pattern presents itself as *query* service with the service name *wiring* and the communication object names $\{$*smartCommWiring, smartCommWiring*$\}$.

The Wiring Slave Figure 5.164 summarizes the internals of the *wiring slave*. The *wiring slave* does not possess any user accessible methods. A service requestor exposes itself as port by calling its *add* and *remove* methods. These then invoke the *internalAdd* and *internalRemove* methods of the *wiring slave* with the appropriate parameters. The first argument of the *internalAdd* method is the *void*-casted *this*-pointer of the service requestor, the next two arguments specify the callback methods used to invoke a *connect/disconnect* on the service requestor and the last argument specifies the name of the port. The returned status code indicates whether the service requestor got registered as port. The *internalRemove* method solely requires the name of the port to be removed.

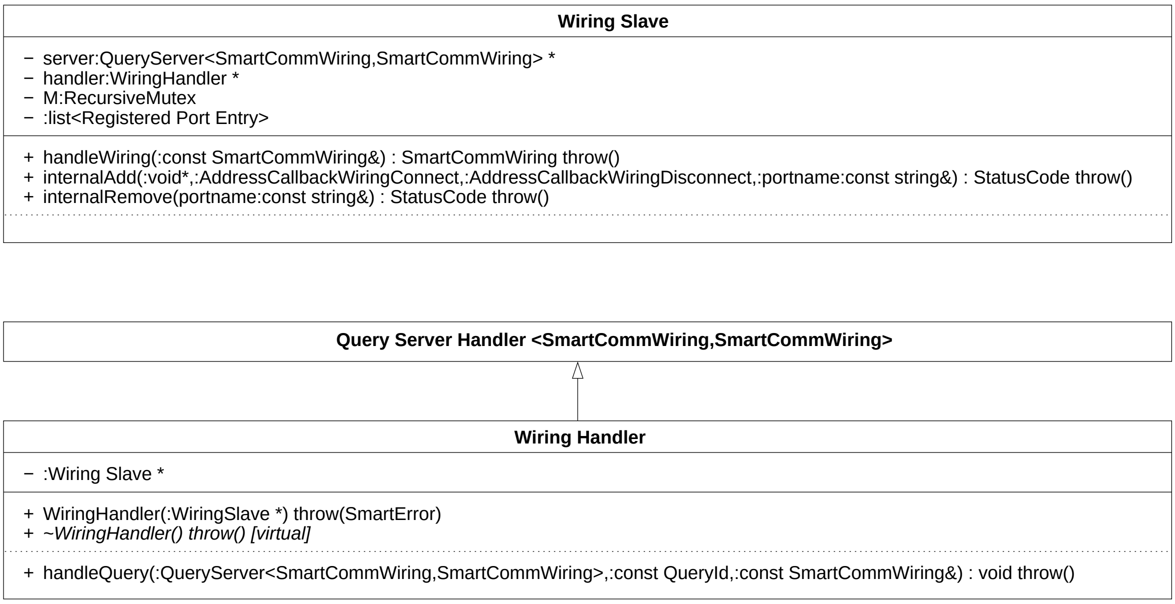

Wiring Slave
− server:QueryServer<SmartCommWiring,SmartCommWiring> *
− handler:WiringHandler *
− M:RecursiveMutex
− :list<Registered Port Entry>
+ handleWiring(:const SmartCommWiring&) : SmartCommWiring throw()
+ internalAdd(:void*,:AddressCallbackWiringConnect,:AddressCallbackWiringDisconnect,:portname:const string&) : StatusCode throw()
+ internalRemove(portname:const string&) : StatusCode throw()
Query Server Handler <SmartCommWiring,SmartCommWiring>
Wiring Handler
− :Wiring Slave *
+ WiringHandler(:WiringSlave *) throw(SmartError)
+ ~WiringHandler() throw() [virtual]
+ handleQuery(:QueryServer<SmartCommWiring,SmartCommWiring>,:const QueryId,:const SmartCommWiring&) : void throw()

Figure 5.164: *Details of the internals of the* wiring slave.

The mutex M protects the list of registered ports and also ensures that a service requestor can not get destroyed while the *wiring slave* executes one of the callback methods of a registered service requestor. The *internalAdd/internalRemove* methods operate in the same way as the *signUp/signOff* methods of the component management class. The mutex M is hold by the *wiring slave* while it stays in one of the callback methods of a service requestor. Thus, the *remove* method can not be executed concurrently and the destruction of the service requestor is delayed in the same way as it is the case with the *signOff* method. The internal representation of registered port is illustrated in figure 5.165.

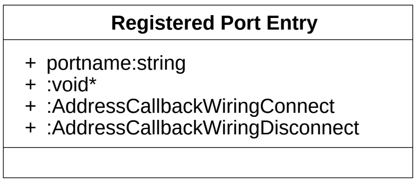

Registered Port Entry
+ portname:string
+ :void*
+ :AddressCallbackWiringConnect
+ :AddressCallbackWiringDisconnect

Figure 5.165: *The representation of a* port *at the* wiring slave*.*

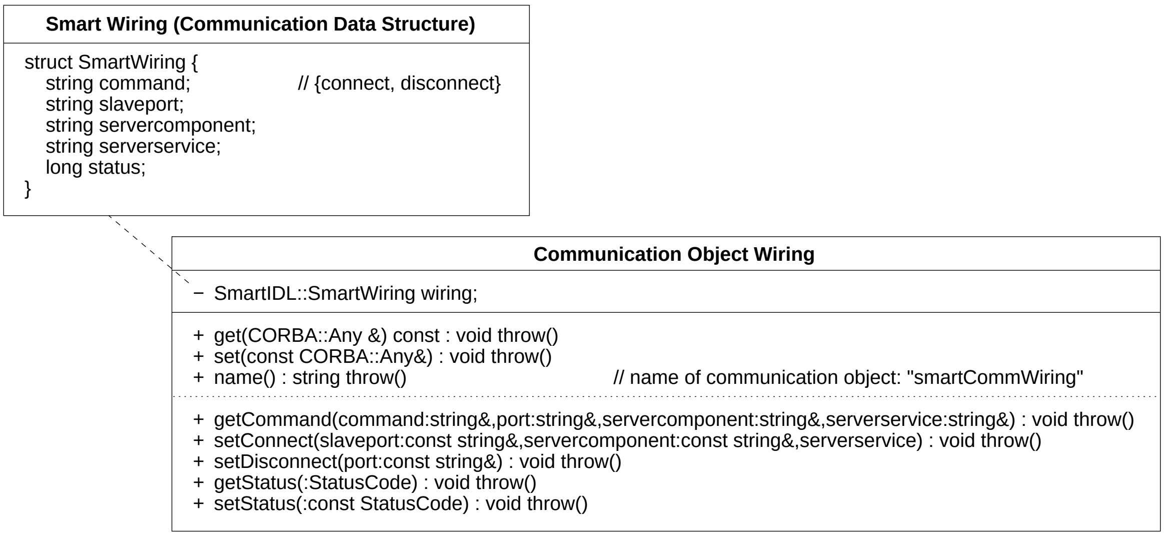

Smart Wiring (Communication Data Structure)
struct SmartWiring {
string command; // {connect, disconnect}
string slaveport;
string servercomponent;
string serverservice;
long status;
}
Communication Object Wiring
− SmartIDL::SmartWiring wiring;
+ get(CORBA::Any &) const : void throw()
+ set(const CORBA::Any&) : void throw()
+ name() : string throw() // name of communication object: "smartCommWiring"
+ getCommand(command:string&,port:string&,servercomponent:string&,serverservice:string&) : void throw()
+ setConnect(slaveport:const string&,servercomponent:const string&,serverservice) : void throw()
+ setDisconnect(port:const string&) : void throw()
+ getStatus(:StatusCode) : void throw()
+ setStatus(:const StatusCode) : void throw()

Figure 5.166: *Details of the communication object used inside the* wiring *pattern.*

The *wiring slave* hosts the server part of the *query* pattern since the *wiring slave* has to implement the instructed wiring. For reasons of simplification, the same communication object is used for the request and the response. As shown in figure 5.166, the specification of the desired wiring consists of the name of the port that is to be wired and the name of the component and service to which the port is to be connected.

The *wiring slave* receives the specification for a new wiring via the *query* pattern. The *query* server invokes the provided handler with each request and the handler invokes the *handleWiring* method of the *wiring slave*. That acquires the mutex M and iterates through the list of ports. In case of a valid port name, either the callback method to perform a *connect* or the one to perform a *disconnect* are invoked on the proper service requestor. The *handleWiring* method returns the status in form of the communication object and the handler of the *query* server invokes the *answer* method of the *query* server to complete the interaction. Finally, the mutex M is released.

The Wiring Master Figure 5.167 summarizes the internals of the *wiring master*. In contrast to the other communication patterns, the *connect* and *disconnect* methods do not connect the *wiring master* with a *wiring slave*. The *connect* method connects the port *slaveport* of the component *slavecmpt* with the service *serversvc* of the component *servercmpt* and the *disconnect* method resolves the connection of the designated port.

The internals of the *connect* method are shown in figure 5.168. First, a new *query client* instance

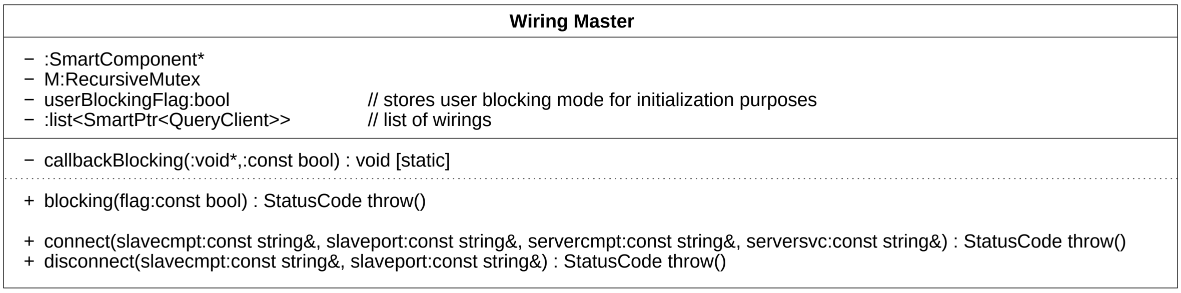

Wiring Master
− :SmartComponent*
− M:RecursiveMutex
− userBlockingFlag:bool // stores user blocking mode for initialization purposes
− :list<SmartPtr<QueryClient>> // list of wirings
− callbackBlocking(:void*,:const bool) : void [static]
+ blocking(flag:const bool) : StatusCode throw()
+ connect(slavecmpt:const string&, slaveport:const string&, servercmpt:const string&, serversvc:const string&) : StatusCode throw()
+ disconnect(slavecmpt:const string&, slaveport:const string&) : StatusCode throw()

Figure 5.167: *Details of the* wiring master.

is created that is added to the *list of wirings*. This allows the *blocking* method of the *wiring master* to invoke the *blocking* method of the *query clients*. Next, the *query client* connects to the *wiring slave* of the component *slavecmpt* by using the reserved service name *wiring*. The interaction with the *wiring slave* is handled by means of a blocking *query*. Finally, the *query client* is destroyed. The *disconnect* method works in the same way and also creates its own *query client* instance. Thus, there can be any number of concurrent method invocations. It has to be noticed that the handler at the *wiring slave* always serializes all concurrent requests. The finally active configuration depends on the order the wiring requests at the *wiring slave*.

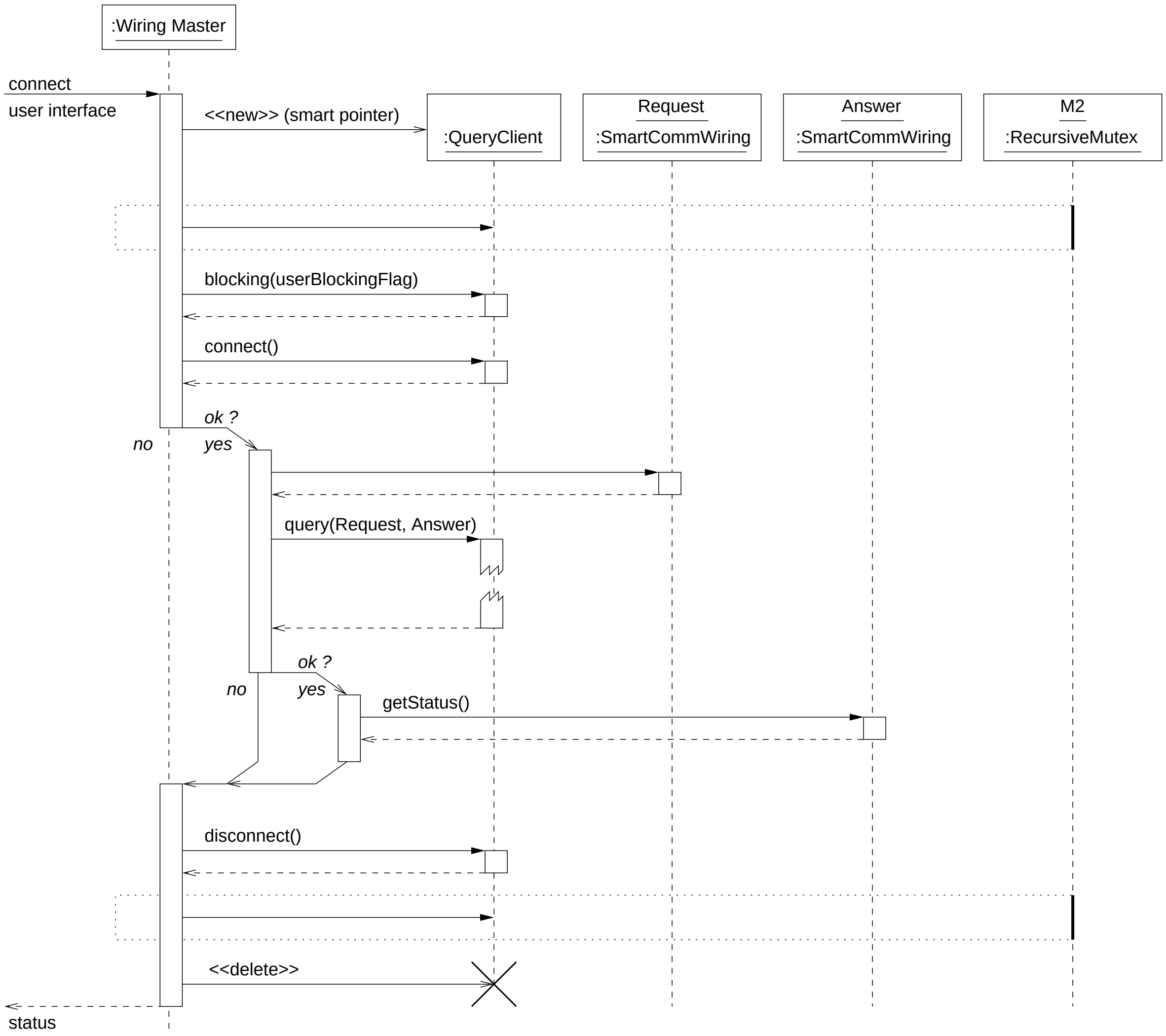


Figure 5.168: *The internals of the* connect *method of the* wiring master.

5.7 The Application Builder View on the Approach

Figure 5.169 shows the application builder view on the approach. The externally visible component interfaces are all composed of the standard communication patterns that are typed by the used communication objects. Thus, one can easily recognize what type of service a component provides, how it is used and what kind of services are required. The component internal dependencies of services are only illustrated in case restrictions apply due to the used types of handlers. As can be seen from the graphical representation, even a tailback can not cause a deadlock in the example setting since there are no circular dependencies.

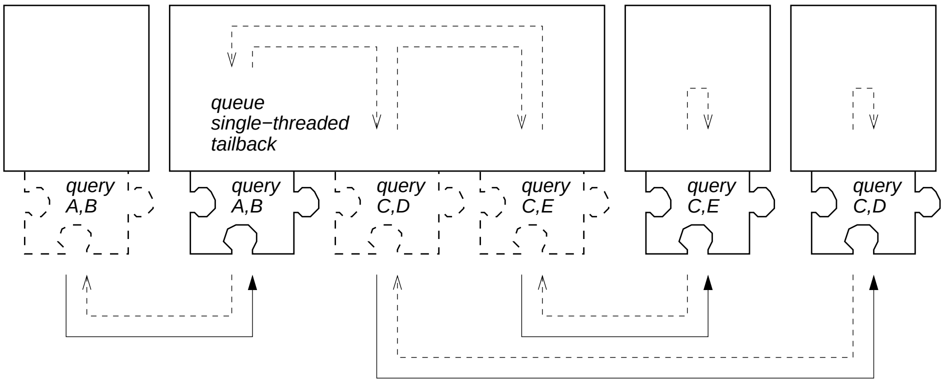


Figure 5.169: *The application builder view on the approach.*

5.8 Additional Framework Infrastructure

5.8.1 The State Management Pattern

The *state* pattern supports a master-slave relationship to selectively activate and deactivate states. An activity can lock a state at the *slave* to inhibit state changes at critical sections as shown in figure 5.170. This prevents an activity from being interrupted at an unsuitable point of execution. The *state* pattern gives the *master* precedence for state changes over the *slave*. As soon as a request for a state change is received from the *master*, the *slave* rejects locks for states that are not compatible to the pending state change of the *master*. The requested state change of the *master* is executed by the *slave* as soon as all locks for states affected by the state change are released. The *state* pattern is, for example, used by the task monitoring component of the sequencing layer for graceful deactivation of component internal user activities. User provided handlers at the *slave* allow for cleanup tasks with a state change.

Figure 5.171 shows an example of a state configuration. A *main state* can comprise several *substates*. At the *slave*, only substates can be locked and the *master* can only set a main state. At each point of time, there can be only one main state active but an arbitrary number of substates. At startup of the *slave*, the main state *neutral* is defined by default and all other states are user defined. Each user defined main state comprises the substate *nonneutral* by default. This allows to start activities as soon as the main state *neutral* is left and to stop them as soon as it is entered.

In case of a requested state change, substates that are not covered by the next main state can not be locked anymore. As soon as all locks are released of substates that get invalid, the state change is performed. At first, the *quitHandler* is invoked with all substates that are not anymore part of the next

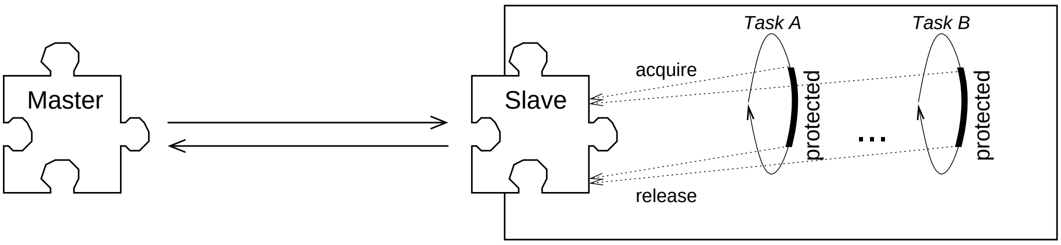


Figure 5.170: *The* state *pattern.*

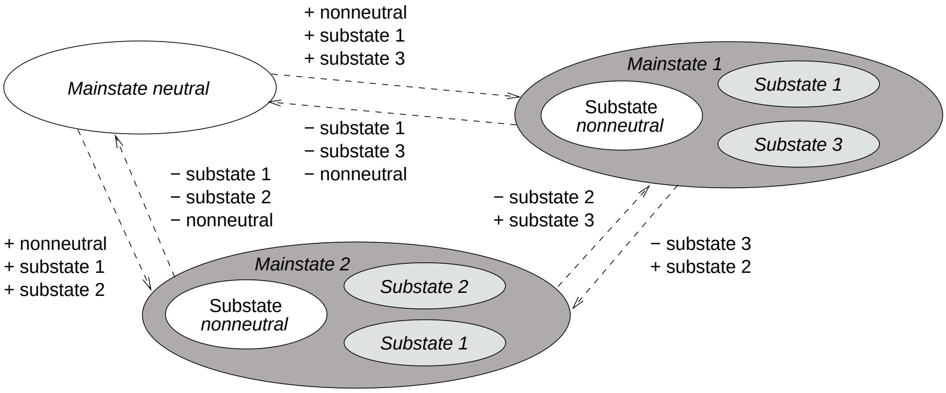


Figure 5.171: *States in white ellipses are provided by default. The other states are defined by the component with startup.*

main state. Then, the *enterHandler* is invoked for all substates that get newly activated with the next main state. In case the substate *nonneutral* is affected, the *quitHandler* is invoked last on it and the *enterHandler* first. This simplifies component internal housekeeping activities.

Requesting the main state *neutral* causes the *slave* to set the component blocking mode to *false* until all locks of substates are released. The effect is that all blocking methods of communication patterns are cancelled so that the substates get released very quickly and independently of the response times of other components.

The class diagram of the *state* pattern is shown in figure 5.172. It is internally based on the *query* pattern. The states can be defined by the *slave* only prior to activating the *slave*. Afterwards, a *master* can get connected to either perform state changes or to ask query the state configuration.

The distinction of *main states* and *substates* prevents from activating incompatible subsets of substates. Of course, in rare cases, one needs all possible combinations of substates which results in a high number of main states. However, components typically comprise only a small number of substates.

5.8.2 The Trader Service

The *trader service* is a stand alone component at which service providers can enroll and where other components can look up services of a specific type. Both interfaces of the *trader service* are based on the *query* pattern.

A service provider enrolls by providing its *component name* and its *service name*. That is the

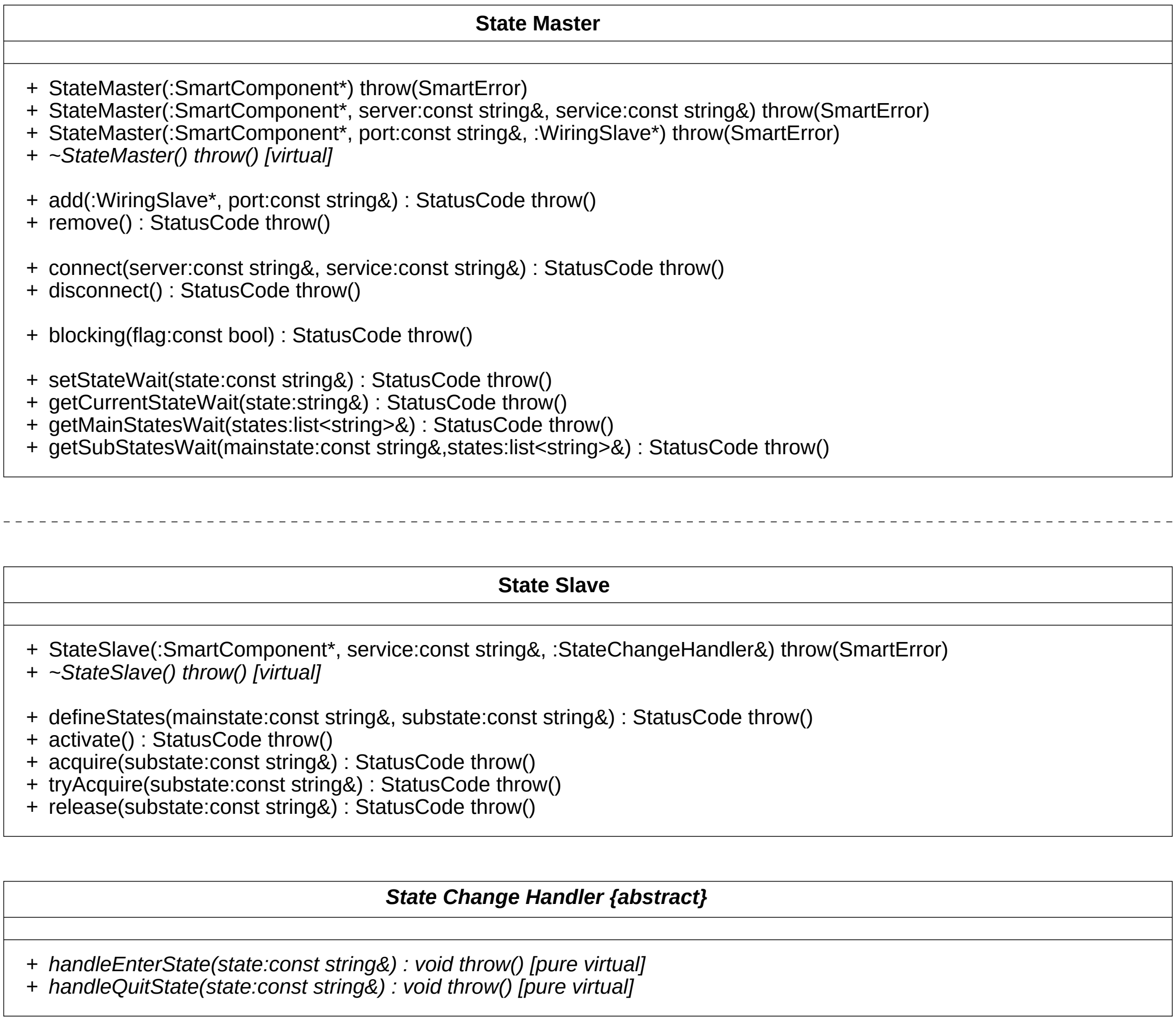

State Master
+ StateMaster(:SmartComponent*) throw(SmartError)
+ StateMaster(:SmartComponent*, server:const string&, service:const string&) throw(SmartError)
+ StateMaster(:SmartComponent*, port:const string&, :WiringSlave*) throw(SmartError)
+ ~StateMaster() throw() [virtual]
+ add(:WiringSlave*, port:const string&) : StatusCode throw()
+ remove() : StatusCode throw()
+ connect(server:const string&, service:const string&) : StatusCode throw()
+ disconnect() : StatusCode throw()
+ blocking(flag:const bool) : StatusCode throw()
+ setStateWait(state:const string&) : StatusCode throw()
+ getCurrentStateWait(state:string&) : StatusCode throw()
+ getMainStatesWait(states:list<string>&) : StatusCode throw()
+ getSubStatesWait(mainstate:const string&,states:list<string>&) : StatusCode throw()
State Slave
+ StateSlave(:SmartComponent*, service:const string&, :StateChangeHandler&) throw(SmartError)
+ ~StateSlave() throw() [virtual]
+ defineStates(mainstate:const string&, substate:const string&) : StatusCode throw()
+ activate() : StatusCode throw()
+ acquire(substate:const string&) : StatusCode throw()
+ tryAcquire(substate:const string&) : StatusCode throw()
+ release(substate:const string&) : StatusCode throw()
State Change Handler {abstract}
+ handleEnterState(state:const string&) : void throw() [pure virtual]
+ handleQuitState(state:const string&) : void throw() [pure virtual]

Figure 5.172: *The class diagram of the* state *pattern.*

unique address used by the *connect* method of a service requestor to access the service provider. Furthermore, it provides the *pattern type* and either zero, one or two *communication object types* depending on the *pattern type*. The *pattern type* specifies the type of the communication pattern used by the service provider and is either *send*, *query*, *push newest*, *push timed* or *event*. The *communication object type* is given by the name of the used communication object types as provided by *name* method of the framework interface of each communication object. With respect to the *trader service*, the *wiring* pattern and the *state* pattern are considered as communication patterns without communication objects.

The *trader service* can be asked for service providers matching a particular type of service. The desired service is specified by the *pattern type* and the appropriate number of *communication object types*. The trader service returns a list of {name of component, name of service} tuples denoting all compatible service providers that are known.

5.9 Conclusion

Modern architectures for taskable robots like three-layer architectures impose high demands on the software architecture to become implementable. Without an appropriate software concept, one rarely is able to exploit the full power of these architectures. Often, a missing software concept is identified as the bottleneck in implementing a taskable robot and in extending the capabilities of such a system.

Foremost, the encountered difficulties are related to the complexity issue. Taskable robots are based on a considerable set of different components which require context dependent interactions. Besides the complexity of interacting components, robotics components are in itself already a challenge due to the great variety of different algorithms needed to implement the various skills of a taskable robot.

A general approach to master the complexity issue is the concept of *decoupling* by drawing clear boundaries between components. The presented approach dictates no rules for decomposition but provides support for clear arrangements of component interfaces. This is achieved by standardized communication patterns that provide a predefined semantics for component interactions. Moving access modes from the user domain to a predefined set of communication patterns ensures decoupling and uniform behavior of component interfaces without restricting the component developer in its component internal architecture.

Conducting all component interactions by communication patterns is the key to *dynamic wiring* of components. *Dynamic wiring* can be seen as *the* pattern of robotics to make the data flow and the control flow configurable from outside a component at runtime.

The communication patterns provide an abstraction from the underlying communication system and achieve the same level of decoupling and the same interface semantics independently of the capabilities of the used communication mechanism. The communication patterns have been used on many different communication systems ranging from *TCP* sockets over a message based system to the current *CORBA* based release without changing the interface semantics.

From the component builder view, the separation of communication and algorithms together with a strict interface semantics significantly reduces the complexity of implementing a component since the types of external interactions are already predefined. The component builder cannot circumvent the decoupling concept of the communication patterns.

From the application builder view, the communication patterns simplify reuse and replacement of components and ensure interoperability by avoiding unwanted and unmanageable side effects in component interactions. Software for complex sensorimotor systems can be assembled of approved software components in a puzzle-like manner which significantly reduces the overall system complexity. Furthermore, this avoids starting from scratch with every new application or system and results in increased robustness.

Of course, a framework can always only focus on some aspects. For example, the described structures of the communication patterns are not designed with respect to hard realtime capabilities or quality of service guarantees. The major focus is on *decoupling* while still adhering to specifics of the robotics domain. For example, the level of transparency is adjusted to the need of robotics.

In general, the presented approach assists in building and using distributed and loosely coupled components. It has been successfully applied in several large scale robotic projects ranging from the collaborative research project *SFB 527: Integration of Symbolic and Subsymbolic Mechanisms for Sensorimotor Systems* at the University of Ulm to the industry leaded consortium *MORPHA: The Interaction, Communication and Cooperation between Humans and Intelligent Robot Assistants* and now forms a major part of the robotics open source initiative *OROCOS*.

Chapter 6

Demos and Applications

6.1 The Robotics Evaluation Area

The robotics evaluation area at the basement of *FAW* is shown in figure 6.1. Its configuration refers to the scenario of the *SFB 527 (Integration of Symbolic and Subsymbolic Information Processing in Adaptive Sensorimotor Systems*). It reproduces an indoor environment consisting of several rooms. Their topological configuration allows to choose between alternative paths. The rooms are equipped with different kinds of furnitures like tables and shelves. Objects can be placed on tables and into shelves. The overall configuration of the objects is foremost engineered with respect to the vision capabilities of the robot and to allow grasping even with the simple manipulator of the *RWI B21* platform.

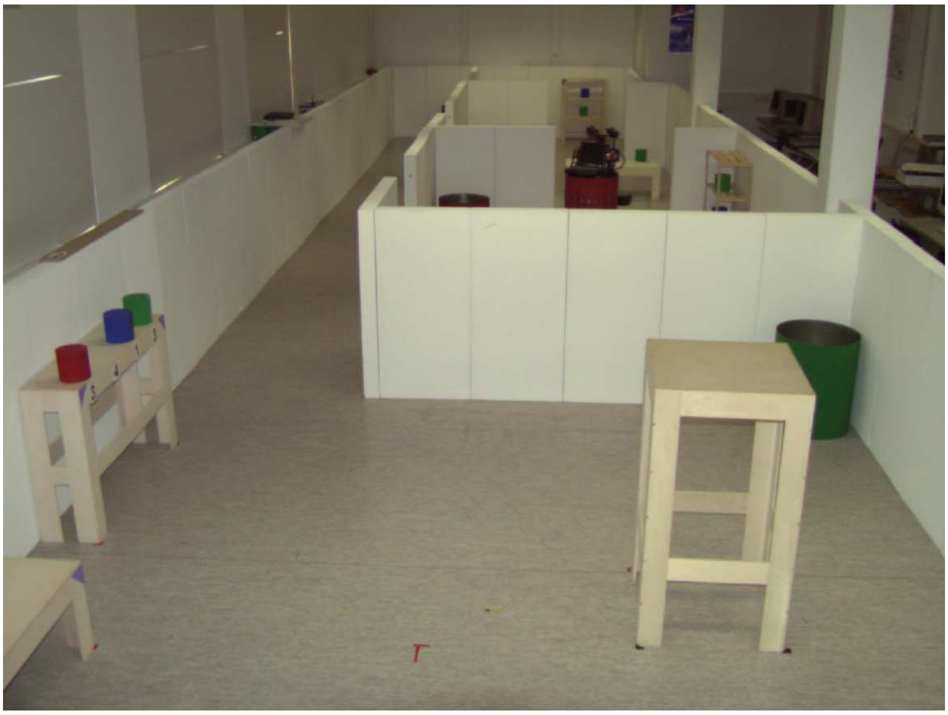


Figure 6.1: *The robotics evaluation area in the basement of the* FAW.

The robotics evaluation area is not only used to evaluate single components but foremost to evaluate their ordered cooperation. Executing pick-and-place tasks requires a substantial amount of steps to get accomplished. In particular, changes in the environment during task execution have an impact on the further task execution plot. Successfully executing tasks in the evaluation area not only requires a substantial amount of skills, but also their dynamic wiring and configuration to form the desired behaviors. Figure 6.2 lists most of the components that are available on the *RWI B21* platform. The implementation of all components is based on the SMARTSOFT framework.

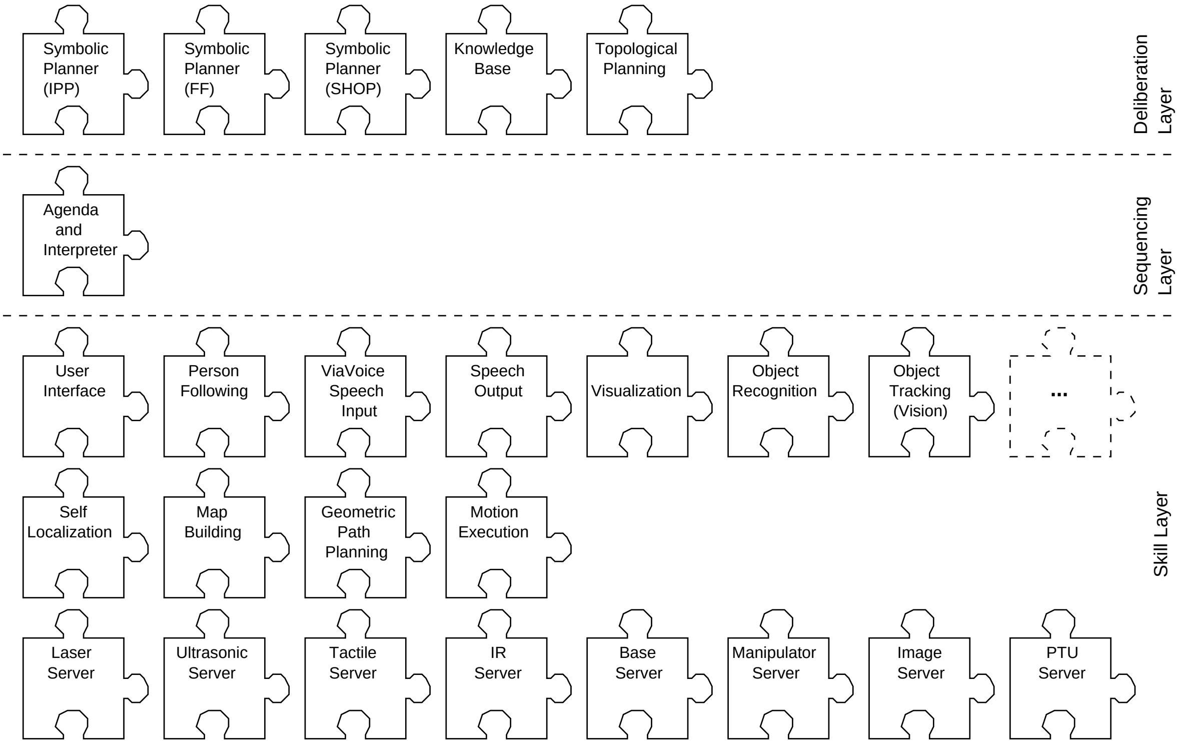


Figure 6.2: *Survey on the installed components.*

6.2 The Pattern Building Task

The *pattern building* task makes up a certain pattern of discs on one of the tables in the evaluation area. In the example, the intentional setting of discs from left to right is *yellow, red, blue, green*. The state of the knowledge base when starting the task execution is shown in figure 6.3. The coordinates of the objects are those perceived by the robot and are maintained and updated in the knowledge base. The pattern of discs on the table in the foreground must not be destroyed. The robot assumes that the shelf in the foreground on the right contains a yellow and a blue disc and that the second shelf on the right in the background contains a red disc in its lowest tray. All other trays of shelves or tables are either assumed unknown or empty which is not distinguished in the visualization.

In the following, images are referred to by *(row/column)*. Figure 6.4 shows the first part of a pattern building task. The overall execution time of the pattern building task is approximately 7 minutes and the robot drives with speeds of up to 1 m/s. During task execution, it is allowed to

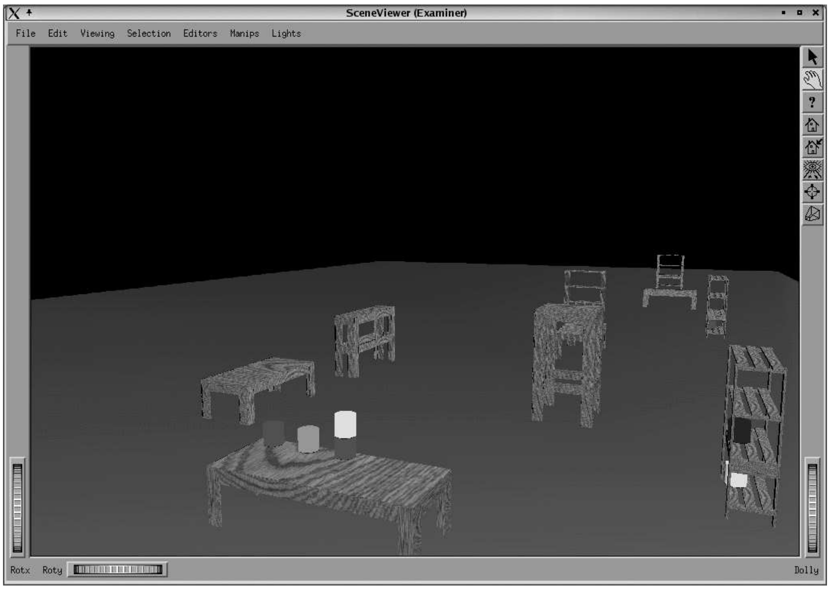


Figure 6.3: *The state of the knowledge base when starting the example task.*

interfere with the task execution by, for example, modifying the configuration of the discs, blocking passages or by constituting a dynamic obstacle.

The robot first moves to the table where the pattern is to be built (1/1-1/2). It then uses its vision system to recognize the current occupation of the table (1/3-2/2). In the task net description of the plot, using the vision system to recognize the current occupation is an action like any other. It can be applied only if the robot is placed correctly and its result is an update of the content of the knowledge base. That includes an update of the table and of the objects located on it including the pose of the table and all perceived discs. Next, the symbolic planner is invoked. The currently available discs are of the correct color so that solely the missing one has to be added. Due to its current state of the knowledge base, the robot assumes there is a red disc in the shelf of room 2. Thus, the result of the symbolic planning step is to move to the shelf, pick the disc, move back to the table and place it there. The plan is transformed in a task-net to get executed. The first step is to move to the shelf (2/3-4/1) and to deploy the manipulator (4/2). The robot then uses the vision system to acknowledge its assumptions on the state of the shelf including updated positions (4/3-5/1). Thereto, the arm is moved downwards to not interact with the field of vision of the cameras. In the example, both trays contain a red disc. The configuration does not match the expected one. However, the symbolic plan only asks the robot to get a red disc from that shelf and leaves it to the sequencing layer to select one. The robot selects the disc in the uppermost tray and grasps it (5/2-5/3). The shelf is approached by using the approaching maneuver of the motion control that is able to simultaneously consider two robot shapes. Otherwise, the robot would not be able to approach the shelf since its contour would collide with the shelf. Figure 6.5 shows the continuation of the grasping maneuver (1/1-1/2). The robot now moves to the table taking into consideration its extended contour due to the payload and the deployed manipulator (1/3-3/1). During movements, the manipulator is moved into the upmost position to not interfere with the field of view of the laser range finder. The vision system is used to check the current table configuration (3/2-4/1) and the manipulator is again moved down. Finally, the manipulator is again moved upwards to let the laser range finder have an unoccluded field of view and when approaching the table to place the disc (4/2-5/1). The approaching maneuver again uses the two-shape configuration.

All the constraints on the various steps are represented in a descriptive way in the task net descriptions of the sequencing layer and are checked by the task net interpreter at execution time. Without a descriptive representation, one would hardly be able to handle such complex interactions.

6.3 The Mail Distribution Task

The *mail distribution* task distributes discs from a *mail shelf* to the shelves of the denoted rooms. As can be seen in (2/2) in figure 6.6, each bin is labeled by a convertible room number. The overall sequence takes approximately 10 minutes to execute. The robot first checks the mail shelf, selects a disc and puts it on a free tray of the shelf in the appropriate room. In case the shelf is full, it reports a task execution failure.

Image (3/3) in figure 6.7 shows another type of detected execution flaw. The laser range finder acknowledged the mail shelf but the vision system was not able to detect it. The reported status of the vision system makes it impossible to the task execution unit to proceed as planned and it thus looks for an alternative plan expansion. In case the knowledge base reports a mailshelf and the vision system reports a detection flaw, a plan expansion applies that asks the user to reposition the robot. The user can decide on whether to continue after a manual repositioning or to abort the overall task. In case of continuing, the task execution unit resumes the original plan expansion so that the vision operator is retried.

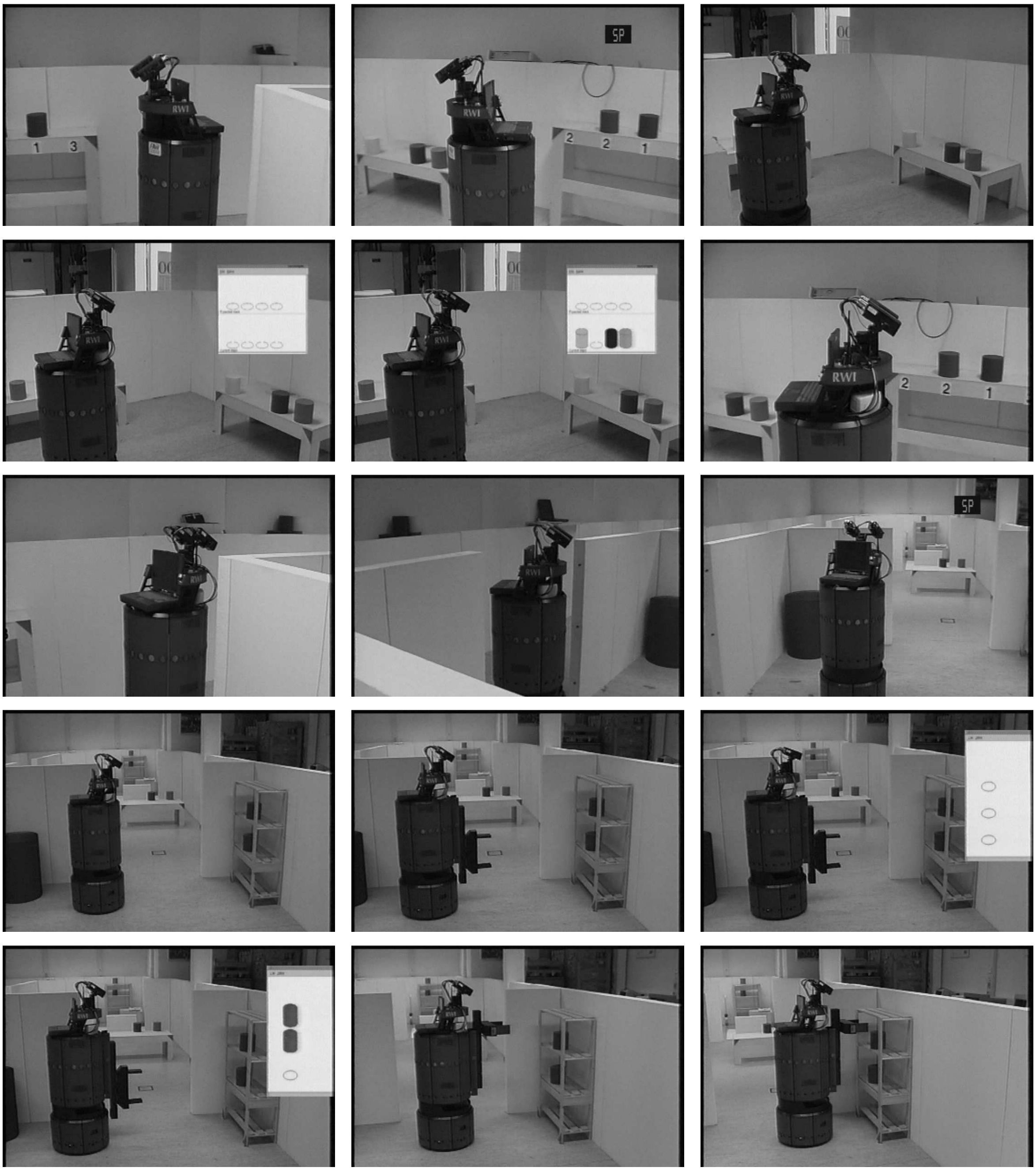


Figure 6.4: *Pattern building task — part one.*

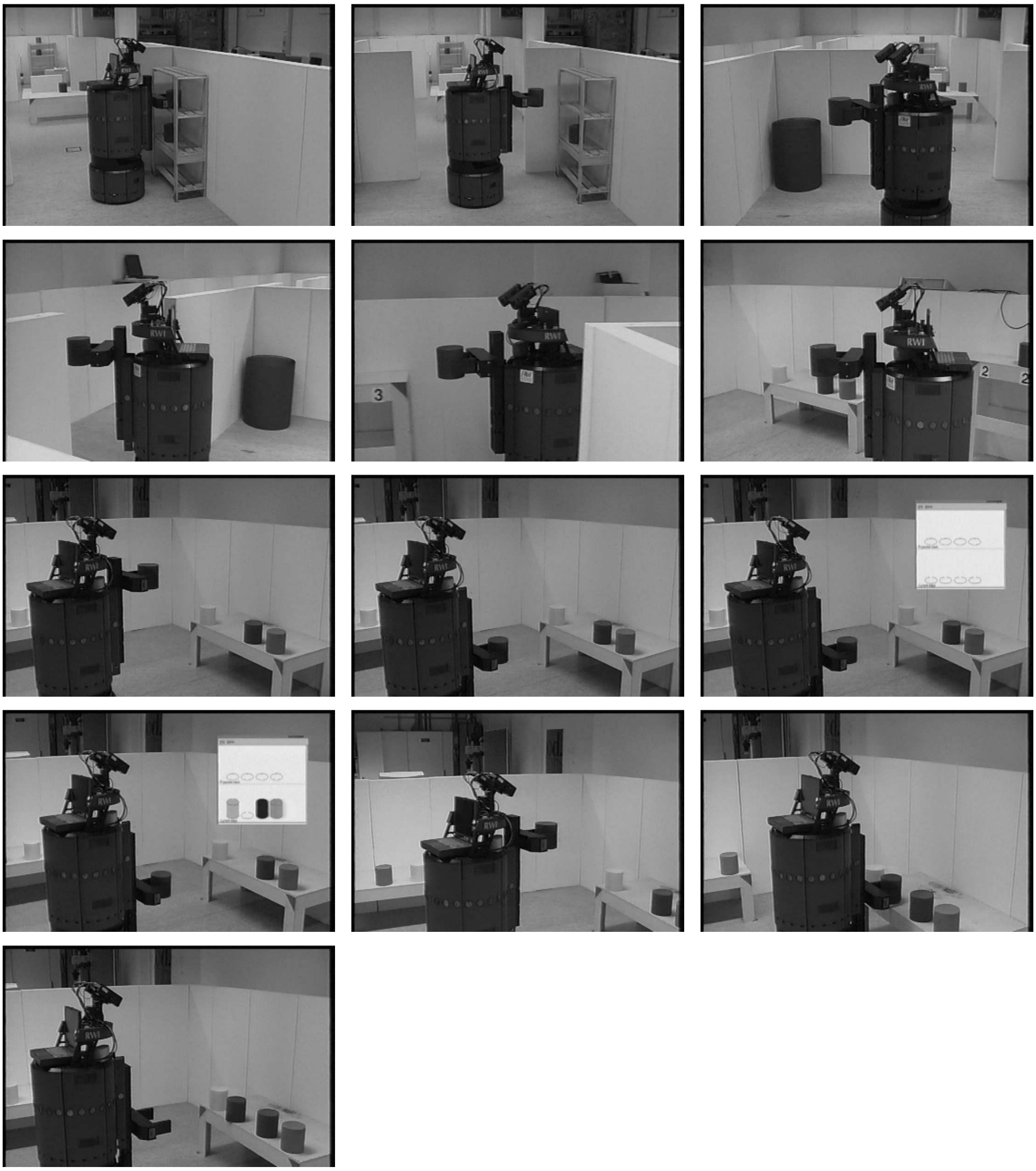


***Figure 6.5:** Pattern building task — part two.*

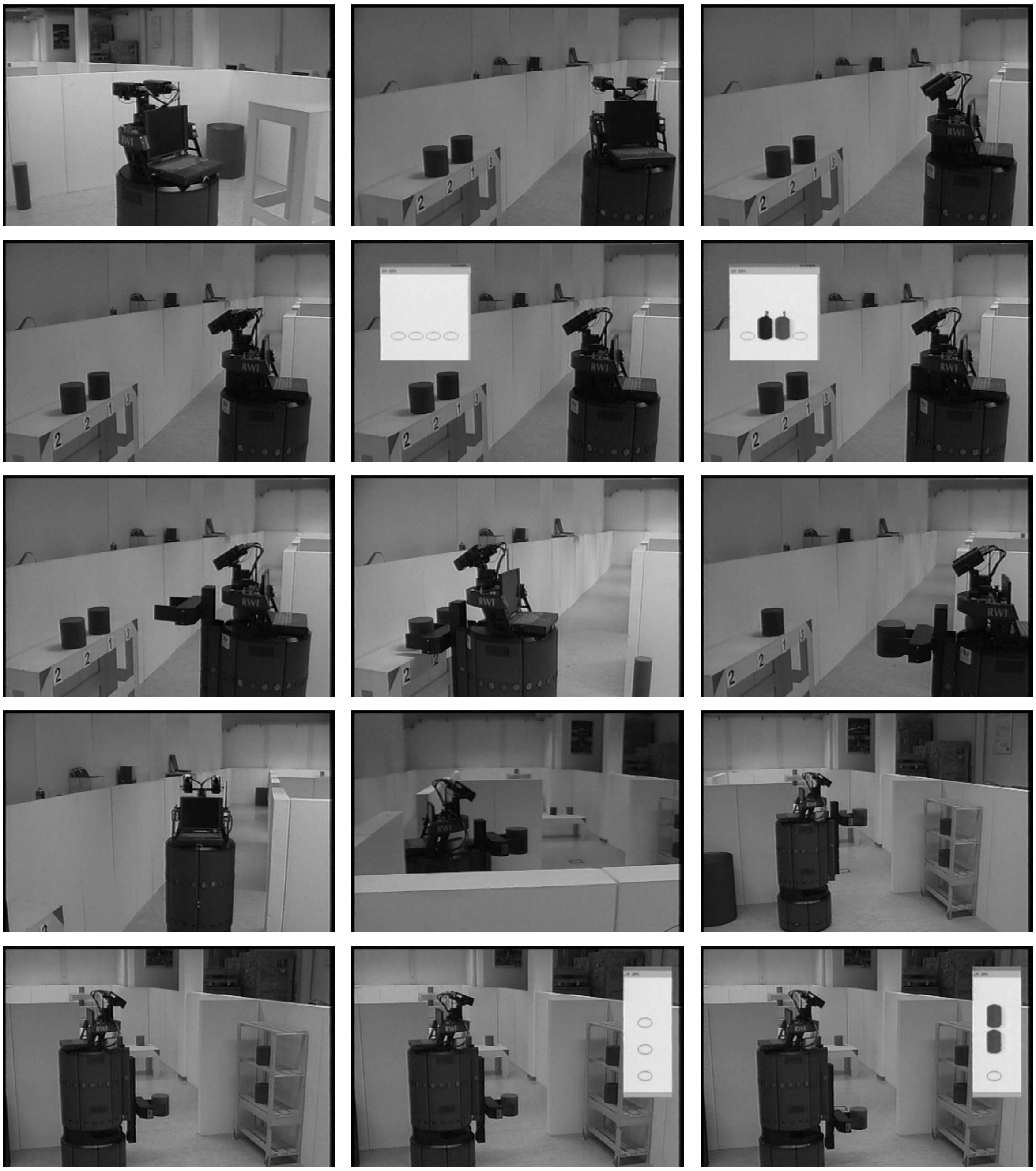


Figure 6.6: *Mail distribution task — part one.*

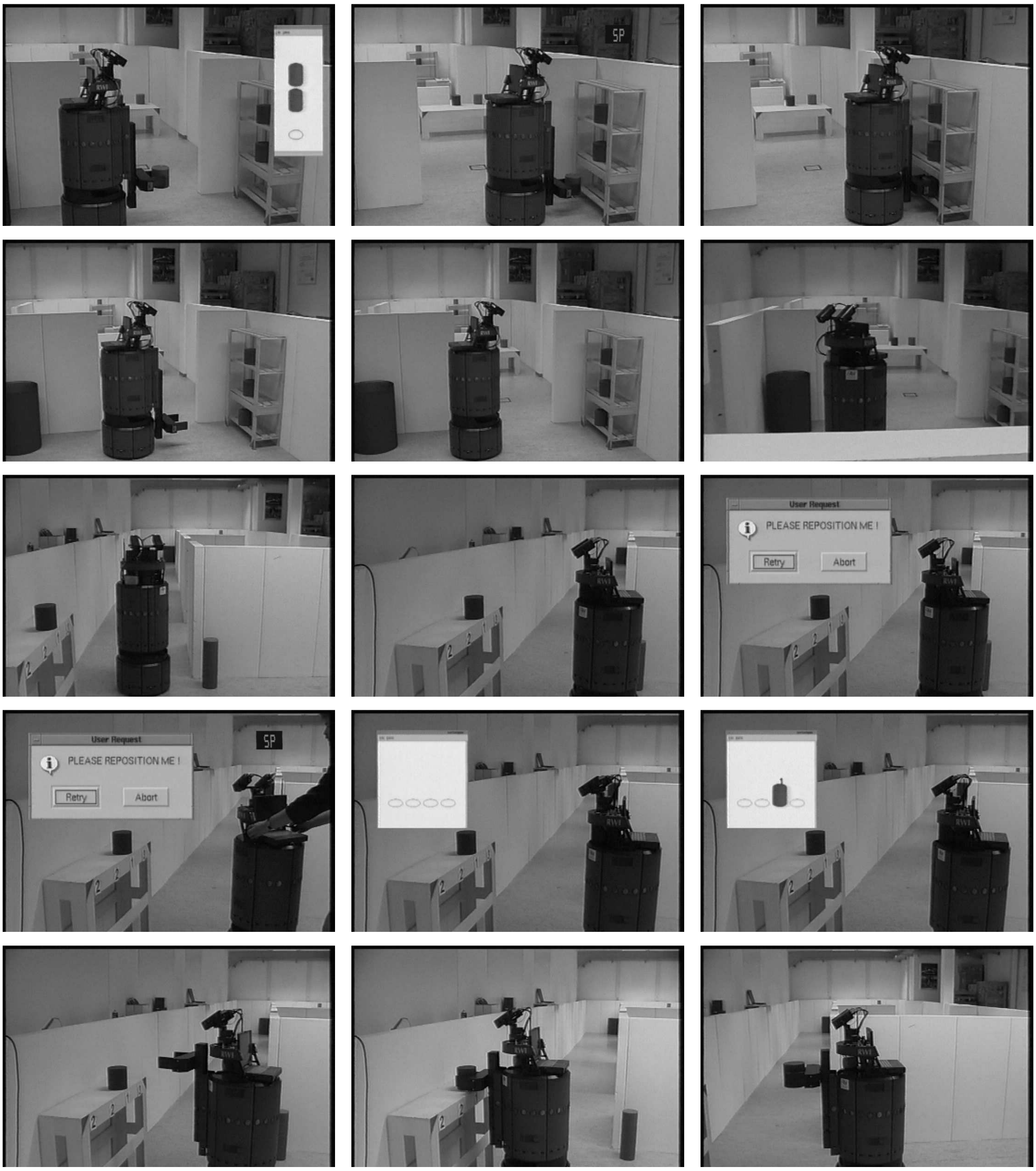


Figure 6.7: *Mail distribution task — part two.*

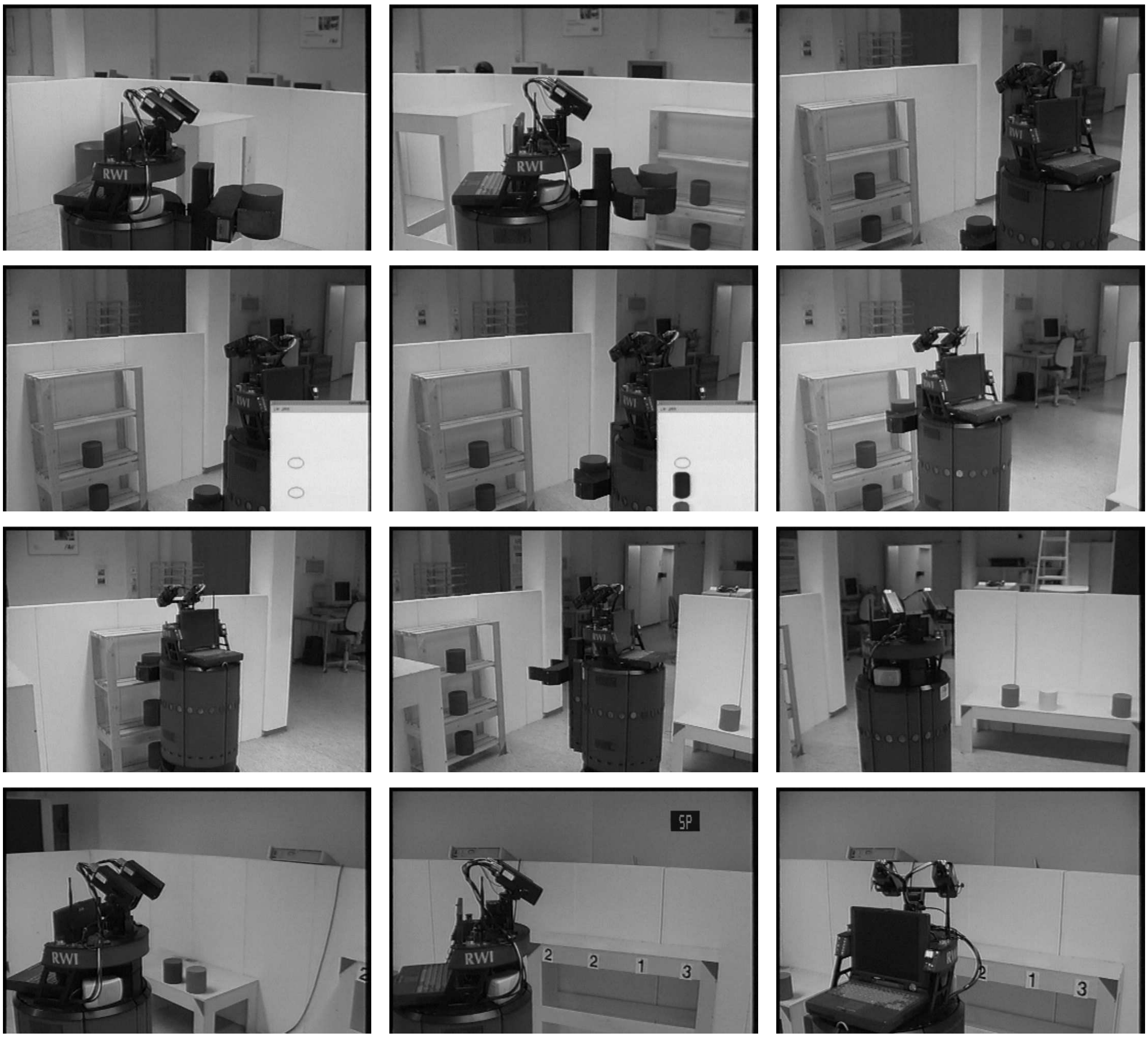


Figure 6.8: *Mail distribution task — part three.*

Chapter 7

Summary and Future Work

7.1 Summary of Results

As soon as robotic systems have to operate in the same environment as humans or even along with humans, they have to perceive the environment and have to react to perceived changes. In reference to the best available knowledge, performing anything more than toy problems requires the integrated use of symbolic and subsymbolic mechanisms of information processing.

Not only that components of such a robotic system are of considerable complexity, overall system complexity is further increased by component interactions. Further demands are imposed by modern architectures for taskable robots like three-layer architectures which require the various skills to be composable at runtime to form behaviors adjusted to the current task and the perceived situation. The situation dependent composition and selection of skills cannot be omitted since it is the only way to execute complex tasks in a dynamic environment. It is very unlikely that a single approach can handle all situations and contingencies experienced in real world. Thus, situation and task dependent composition of behaviors is a poweful approach to level out shortcomings of approaches.

A software concept tailored to the needs of taskable robots has been developed and implemented. The approach dictates no rules for decomposition but provides support for clear arrangements of component interfaces. This is achieved by standardized communication patterns that provide a predefined semantics for component interactions. Conducting all component interactions by communication patterns is the key to the *dynamic wiring* pattern. It can be seen as *the* pattern of robotics since it makes the control flow and the data flow configurable from outside a component at runtime.

The software architecture has been successfully used within several large scale projects ranging from academic to industrial collaborations and demonstrated its capability to master the complexity issue and now forms a major part of the *OROCOS* initiative.

Furthermore, an approach for navigating in dynamic environments has been developed which is balanced in terms of necessary computing power, achieved reactivity and deviation from optimality and completeness and which allows to drive with considerably high speed. It is able to cope with any-shaped robots at the level of motion control and ensures mobility in most situations in dynamic environments. The responsibilities are assigned to the involved components such that the robot always operates in a safe state taking into account its kinematic, dynamic and shape constraints.

Finally, an outline of a computation scheme for simultaneous localization and mapping has been presented [129]. The computation scheme allows to acquire a preliminary map of the environment that can already be used for localization and navigation. Due to a conservative filter, one never gets overly confident in the pose estimate while still being able to extract absolute node poses suitable for exploration by a relaxation based scheme. Since a relative map is used, no information is lost during the exploration phase even that a conservative filter is applied to extract absolute node poses. Thus, the time-consuming global optimization can be postponed without loss of information.

A full-fledged and taskable mobile robot that is able to perform various complex fetch and carry tasks as well as recognition tasks in a dynamic test environment has been implemented and used to verify the workability of the approaches. It proved that the overall concept of integration as well as the methods used within the various components are one way to accomplish complex tasks under real world constraints.

7.2 Future Work

This thesis only shortly strived the sequencing layer and the interaction with the deliberative layer [130] [128]. However, it proved to be crucial to be able to store procedural knowledge on how to perform a task in a declarative way within task nets. The declarative representation allows to handle all the various details about different settings in a way that is manageable and extensible. The powerful sequencing mechanism is a key feature.

Reliable and stable basic skills combined with a concept for integration are a step forward towards more flexible systems. Even though the system implemented on the algorithms and concepts presented in this thesis is already able to perform tasks of remarkable complexity even in a dynamic environment, it still is only a small step towards autonomous robots.

A lot of procedural knowledge is encoded by means of task nets. For example, this includes knowledge about when and how to apply the vision system. This is the key to be able to provide context dependent knowledge like the expected objects or the height of a table which is needed by the vision based object recognition to determine the positions of objects. However, the symbolic task planner is integrated in the same way. On the one hand, that has the advantage that one can reduce the search space dramatically by providing only that part of the knowledge base that is relevant for the current planning task. On the other hand, that prevents the deliberative layer from continuously monitoring the activities of the sequencing layer.

In general, both situation recognition and the better integration of a deliberative component seem to be decisive on the way towards fully autonomous systems. At present, far too much knowledge on how to decide in a specific situation needs to be provided beforehand. However, only fully integrated systems equipped with reliable and stable skills and a maintainable architecture provide the basis for addressing these issues since systems operating in real world do not allow for unrealistic simplifications.

Bibliography

[1] ActiveX, Microsoft Component Object Model (COM) technologies.
URL *http://www.microsoft.com/com/*.

[2] ActivMedia.
URL *http://robots.activmedia.com/*.

[3] R. C. Arkin. Integrating behavioral, perceptual and world knowledge in reactive navigation. *Robotics and Autonomous Systems*, 6:105–122, 1990.

[4] K. O. Arras, J. Persson, N. Tomatis, and R. Siegwart. Real-time obstacle avoidance for polygonal robots with a reduced dynamic window. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pages 3050–3055. Washington, DC, May 2002.

[5] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons. The design and performance of a scalable ORB architecture for CORBA asynchronous messaging. In *Middleware*, pages 208–230, 2000.

[6] M. Beetz. Structured reactive controllers. *Autonomous Agents and Multi-agent Systems*, 4(1-2): 25–55, 2001.

[7] O. Bengtsson and A.-J. Baerveldt. Robot Localization based on Scan-Matching - Estimating the Covariance Matrix for the IDC Algorithm. *Robotics and Autonomous Systems*, 44(1):29–40, 2003.

[8] R. P. Bonasso. Integrating reaction plans and layered competences through synchronous control. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 1225–1231, 1991.

[9] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Computer Science*, 9(2):237–256, 1997.

[10] R. P. Bonasso and D. Kortenkamp. Characterizing an architecture for intelligent, reactive agents. In *Working Notes, AAAI Spring Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*, pages 29–34, 1995.

[11] The Boost Smart Pointer Library.
URL *http://www.boost.org/*.

[12] J. Borenstein and Y. Koren. The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3):278–288, 1991.

[13] O. Brock. *Generating Robot Motion: The Integration of Planning and Execution.* PhD thesis, Stanford University, 2000.

[14] O. Brock and O. Khatib. High-speed navigation using the global dynamic window approach. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 1999.

[15] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[16] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, RA-2(1), March 1986.

[17] R. A. Brooks. The behavior language user's guide. Technical report, Artificial Intelligence Laboratory Memo 1227, Massachusetts Institute of Technology, Cambridge, Mass., 1989.

[18] R. A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6:3–15, 1990.

[19] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plásil, G. Pomberger, W. Pree, and C. Szyperski. What characterizes a (software) component ? *Software Concepts & Tools*, 19(1), 1998.

[20] H. Bruyninckx. Open robot control software: The OROCOS project. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pages 2523–2528. Seoul, Korea, May 2001.

[21] J. Canny and J. Reif. New lower bound techniques for robot motion planning problems. In *Proc. of the 27th Annual IEEE Symposium on the Foundations of Computer Science*, pages 49–60. Los Angeles, 1987.

[22] J. A. Castellanos, J. D. Tardós, and et al. Building a global map of the environment of a mobile robot: The importance of correlations. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 1053–1059. Albuquerque, 1997.

[23] K. Chong and L. Kleeman. Mobile robot map building from an advanced sonar array and accurate odometry. *Int. J. Robotics Research*, 18(1):20–36, 1999.

[24] J. Connell. A colony architecture for an artificial creature. Technical Report 1151, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1989.

[25] J. Connell. SSS: A hybrid architecture applied to robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2719–2724, 1992.

[26] CORBA, Object Management Group, Inc. (OMG).
URL *http://www.corba.org/*.

[27] OMG Catalog of CORBA/IIOP Specifications.
URL *http://www.omg.org/technology/documents/corba-spec-catalog.htm*.

[28] Catalog of OMG IDL / Language Mappings Specifications.
URL *http://www.omg.org/technology/documents/idl2x-spec-catalog.htm*.

[29] OMG CORBA 3 release information, september 2002.
URL *http://www.omg.org/technology/corba/corba3releaseinfo.htm*.

[30] CORBA Component Model, Object Management Group, Inc. (OMG).
URL *http://www.omg.org/*.

[31] E. Coste-Manière and R. Simmons. Architecture, the backbone of robotic systems. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pages 67–72, 2000.

[32] M. Csorba. *Simultaneous Localisation and Map Building*. PhD thesis, Robotics Research Group, University of Oxford, 1997.

[33] M. W. M. G. Dissanayake, P. Newman, H. F. Durrant-Whyte, S. Clark, and M. Csorba. A solution to the simultaneous localization and map building (SLAM) problem. Technical report, Australian Centre for Field Robotics, University of Sydney, Australia, 1999. ACFR-TR-01-99.

[34] T. Duckett, S. Marsland, and J. Shapiro. Learning globally consistent maps by relaxation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 4, pages 3841–3846, 2000.

[35] T. Duckett, S. Marsland, and J. Shapiro. Fast online learning of globally consistent maps. *Autonomous Robots*, 12(3):287–300, 2002.

[36] J. R. Ellis and D. L. Detlefs. Efficient Garbage Collection for C++. In *Usenix Proceedings*, February 1994.

[37] C. Elsaessar and M. Slack. Integrating deliberative planning in a robot architecture. In *Proceedings of the American Institute of Aeronautics and Astronautics / NASA Conference on Intelligent Robots in Field, Factory, Service and Space (CIRFFSS)*. NASA Conference Publication 3251, Houston, Texas, 1994.

[38] H. J. S. Feder and J. J. E. Slotina. Real-time path planning using harmonic potentials in dynamic environments. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*. Albuquerque, 1997.

[39] W. Feiten, R. Bauer, and G. Lawitzky. Robust obstacle avoidance in unknown and cramped environments. In *Proc. IEEE International Conference on Robotics and Automation*, pages 2412–1217. San Diego, May 1994.

[40] P. Fiorini and Z. Shiller. Motion planning in dynamic environments using velocity obstacles. *International Journal of Robotics Research*, 17(7):760–772, July 1998.

[41] R. J. Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, 1989.

[42] R. J. Firby. *The xRAP Language Manual*. Intell/Agent Systems, 1840 Oak Avenue, Evanston, IL 60201, 1997.

[43] R. J. Firby, E. R. Kahn, P. N. Prokopowicz, N. Peter, and J. M. Swain. An architecture for vision and action. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 72–79, 1995.

[44] R. J. Firby, P. N. Prokopowicz, J. M. Swain, R. E. Kahn, and D. Franklin. Programming CHIP for the IJCAI-95 robot competition. *AI Magazine*, 17(1):71–81, 1996.

[45] B. Flannery, W. Press, S. Teukolsky, and W. Vetterling. *Numerical Recipes, Second Edition*. Cambridge University Press, 1992.

[46] S. Fleury, M. Herrb, and A. Mallet. GenoM user's manual. Technical Report 01577, LAAS-CNRS, December 2001.
URL *http://www.laas.fr/˜mallet/genom/genom.pdf*.

[47] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1), 1997.

[48] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[49] U. Frese and T. Duckett. A multigrid approach for accelerating relaxation-based SLAM. In *18. Fachtagung Autonome Mobile Systeme (AMS)*, Informatik aktuell, pages 192–202. Springer, Karlsruhe, December 2003.

[50] U. Frese and G. Hirzinger. Simultaneous localization and mapping - a discussion. In *Proceedings of the IJCAI Workshop on Reasoning with Uncertainty in Robotics*. Seattle, 2001.

[51] K. Fujimura. *Motion Planning in Dynamic Environments*. Computer Science Workbench. Springer, 1991.

[52] E. Gat. ALFA: A language for programming reactive robotic control systems. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1116–1121, 1991.

[53] E. Gat. *Reliable Goal-Directed Reactive Control for Real-World Autonomous Mobile Robots*. PhD thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, 1991.

[54] E. Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the IEEE Aerospace Conference*, 1997.

[55] E. Gat and G. Dorais. Robot navigation by conditional sequencing. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1293–1299, 1994.

[56] M. Georgeff and A. Lanskey. Reactive reasoning and planning. In *Proceedings of the Conference of the American Association of Artificial Intelligence*, pages 677–682, 1987.

[57] G. De Giacomo, Y. Lespérance, and H. J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[58] Covariance Intersection Working Group. A culminating advance in the theory and practice of data fusion, filtering and decentralized estimation, 1997.
URL *http://www.ait.nrl.navy.mil/people/uhlmann/CovInt.html*.

[59] J. Guivant and E. Nebot. Optimization of the simultaneous localization and map building algorithm for real time implementation. *IEEE Transactions on Robotics and Automation*, 17 (3):242–257, 2001.

[60] J. Gutmann and K. Konolige. Incremental mapping of large cyclic environments. In *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*, 2000.

[61] J. Gutmann and C. Schlegel. AMOS: Comparison of scan matching approaches for self-localization in indoor environments. In *Proceedings of the 1st Euromicro Workshop on Advanced Mobile Robots (EUROBOT)*, pages 61–67. IEEE Computer Society Press, 1996.

[62] D. Hähnel, W. Burgard, and G. Lakemeyer. GOLEX - bridging the gap between logic (GOLOG) and a real robot. In *German Conference on Artificial Intelligence (KI)*, 1998.

[63] K. Haigh and M. Veloso. High-level planning and low-level execution: Towards a complete robotic agent. In *Proceedings of the First International Conference on Autonomous Agents*. Menlo Park, CA, 1997.

[64] D. Hall and J. Llinas, editors. *Handbook of Multisensor Data Fusion*. The Electrical Engineering and Applied Signal Processing Series. CRC Press, 2001.

[65] U. D. Hanebeck and K. Briechle. New results for stochastic prediction and filtering with unknown correlations. In *Proc. of IEEE Multisensor Fusion and Integration for Intelligent Systems*, 2001.

[66] U. D. Hanebeck, K. Briechle, and J. Horn. A tight bound for the joint covariance of two random vectors with unknown but constrained cross-correlation. In *Proc. of IEEE Multisensor Fusion and Integration for Intelligent Systems*, 2001.

[67] R. Hartley and F. Pipitone. Experiments with the subsumption architecture. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 1991.

[68] C. Hartwich. Why it is so difficult to build N-tiered Enterprise Applications. Technical Report B 01-05, Institut für Informatik, Freie Universität Berlin, November 2001.

[69] C. A. R. Hoare. Monitors: An operating system structuring mechanism. *Communications of the ACM*, 17, 1974.

[70] J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[71] R. A. Horn and C. R. Johnson. *Matrix Analysis*. Cambridge University Press, 1991.

[72] W3C Architecture domain: HTTP - Hypertext Transfer Protocol.
URL *http://www.w3.org/Protocols/*.

[73] F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 1998.

[74] iRobot. *Mobility 1.1 Robot Integration Software User's Guide*, 1999.
URL *http://www.irobot.com/*.

[75] iRobot.
URL *http://www.irobot.com/*.

[76] Java, Sun Microsystems, Inc.
URL *http://java.sun.com/*.

[77] Javabeans, Sun Microsystems, Inc.
URL *http://java.sun.com/products/javabeans/*.

[78] P. Jensfelt. *Approaches to mobile robot localization in indoor environments*. PhD thesis, Royal Institute of Technology, Sweden, 2001.

[79] Jini, Sun Microsystems, Inc.
URL *http://wwws.sun.com/software/jini/*.

[80] S. Julier. A sparse weight kalman filter approach to simultaneous localisation and map building. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1251–1256, 2001.

[81] S. Julier. The stability of covariance inflation methods for SLAM. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2003.

[82] S. Julier and J. Uhlmann. A nondivergent estimation algorithm in the presence of unknown correlations. In *Proceedings of the American Control Conference (ACC)*, pages 2369–2373, 1997.

[83] S. Julier and J. Uhlmann. A counter example to the theory of simultaneous localization and map building. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 4238–4243, 2001.

[84] L. P. Kaelbling. REX: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings of the AIAA Conference on Computers in Aerospace*, 1987.

[85] L. P. Kaelbling. Goals as parallel program specifications. In *Proceedings of Seventh National Conference on Artificial Intelligence AAAI*, pages 60–65, 1988.

[86] R. Kalman. A new approach to linear filtering and prediction problems. *ASME Journal Basic Engineering*, 82:34–45, 1960.

[87] L. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration space for fast path planning. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pages 2138–2145, 1994.

[88] M. Khatib and R. Chatila. An extended potential field approach for mobile robot sensor-based motion. In *Intelligent Autonomous Systems (IAS-4)*. IOS Press, 1995.

[89] M. Khatib, H. Jaouni, R. Chatila, and J. P. Laumond. Dynamic path modification for car-like nonholonomic mobile robots. In *Proc. IEEE International Conference on Robotics and Automation*. Albuquerque, 1997.

[90] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Int. Journal of Robotics Research*, 5(1), 1986.

[91] B. Kluge, D. Bank, and E. Prassler. Motion coordination in dynamic environments: Reaching a moving goal while avoiding moving obstacles. In *Proc. of International Workshop on Robot and Human Interactive Communication*. Berlin, 2002.

[92] J. Knight, A. Davison, and I. Reid. Towards constant time SLAM using postponement. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 405–413, 2001.

[93] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proceedings of European Conference on Planning (ECP)*, LNAI 1348, pages 273–285. Springer, 1997.

[94] K. Konolige. COLBERT: A language for reactive control in saphira. *KI - Künstliche Intelligenz*, pages 31–52, 1997.

[95] K. Konolige. *Saphira Robot Control Architecture Saphira Version 8.1.0.* SRI International, April 2002.

[96] K. Konolige, K. L. Myers, E. H. Ruspini, and A. Saffiotti. The Saphira architecture: A design for autonomy. *Journal of experimental & theoretical artificial intelligence: JETAI*, 9(1):215–235, 1997.

[97] Y. Koren and J. Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pages 1398–1404. Sacramento, 1991.

[98] D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors. *Artificial Intelligence and Mobile Robots - Case Studies of Successful Robot Systems*. AAAI Press/The MIT Press, 1998.

[99] J. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, 1991.

[100] J. Leonard and H. Durrant-Whyte. Dynamic map building for an autonomous mobile robot. *The International Journal on Robotics Research*, 1992.

[101] J. Leonard and H. Feder. A computationally efficient method for large scale concurrent mapping and localization. In *Robotics Research: Ninth International Symposium, Utah*, pages 169–176, 2000.

[102] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, April-June 1997.

[103] T. Lozano-Perez. Automatic planning of manipulator transfer movements. *IEEE Transactions on Systems, Man and Cybernetics*, 11(10):681–698, 1981.

[104] F. Lu. *Shape registration using optimization for mobile robot navigation*. PhD thesis, Department of Computer Science, University of Toronto, 1995.

[105] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, 4:333–349, 1997.

[106] F. Lu and E. Milios. Robot pose estimation in unknown environments by matching 2d range scans. *Journal of Intelligent and Robotic Systems*, 18(3):249–275, 1997.

[107] P. Maes. Situated agents can have goals. *Robotics and Autonomous Systems*, 6(1-2):49–70, 1990.

[108] A. Mallet, S. Fleury, and H. Bruyninckx. A specification of generic robotics software components: Future evolutions of GenoM in the OROCOS context. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.

[109] D. McDermott. A reactive plan language. Technical Report YALEU/DCS/RR-864, Yale University, 1991.

[110] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*. AAAI, Edmonton, Canada, 2002.

[111] H. Moravec and A. Elfes. High resolution maps from wide angle sonar. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 116–121, 1985.

[112] M. Muehlhaeuser, editor. *First Workshop on Component-Oriented Programming (WCOP)*. Special Issues in Object-Oriented Programming. dpunkt Verlag, Heidelberg, 1997. ISBN 3-920993-67-5.

[113] D. R. Musser, G. J. Derge, and A. Saini. *The STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, 2001.

[114] D. S. Nau, T. C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[115] P. Newman. *On the structure and solution of the simultaneous localisation and map building problem*. PhD thesis, Australian Centre for Field Robotics, The University of Sydney, 1991.

[116] N. J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufman, San Mateo, 1980.

[117] F. R. Noreils. Integrating error recovery in a mobile robot control system. *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 1990.

[118] A. Orebäck and H. Christensen. Evaluation of architectures for mobile robotics. *Autonomous Robots*, 14:33–49, 2003.

[119] OROCOS - Open Source Robot Control Software.
URL *http://www.orocos.org/*.

[120] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. Levine. Evaluating policies and mechanisms for supporting embedded, real-time applications with CORBA 3.0. In *Sixth IEEE Real Time Technology and Applications Symposium (RTAS)*, May 2000.

[121] I. Pembeci and G. D. Hager. A comparative review of robot programming languages. Technical report, CIRL Lab, 2002.
URL *http://www.cs.jhu.edu/CIRL/publications/pdf/pembeci-review.pdf*.

[122] F. Plásil and M. Stal. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. *Software Concepts & Tools*, 19(1):14–28, 1998.

[123] A. Puder. Objekte ohne Grenzen - Object Request Broker: Funktionsumfang und Standardkonformität. *iX*, 12(12):86–95, 2002.

[124] S. Quinlan and O. Khatib. Elastic bands: Connecting path planning and control. In *Proc. IEEE International Conference on Robotics and Automation*. Atlanta, 1993.

[125] J. K. Rosenblatt and D. W. Payton. A fine-grained alternative to the subsumption architecture. In *Proceedings of the AAAI Stanford Spring Symposium Series*, 1989.

[126] C. Schlegel. Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot. In *Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 594–599. Victoria, Canada, 1998.

[127] C. Schlegel. A Component Approach for Robotics Software: Communication Patterns in the OROCOS Context. In *18. Fachtagung Autonome Mobile Systeme (AMS)*, Informatik aktuell, pages 253–263. Springer, Karlsruhe, December 2003.

[128] C. Schlegel, J. Illmann, H. Jaberg, M. Schuster, and R. Wörz. Integrating vision-based behaviors with an autonomous robot. *VIDERE - Journal of Computer Vision Research*, 1(4), 2000.

[129] C. Schlegel and T. Kämpke. Filter design for simultaneous localization and mapping (SLAM). In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 2737–2742, 2002.

[130] C. Schlegel and R. Wörz. Interfacing different layers of a multilayer architecture for sensorimotor systems using the object-oriented framework SMARTSOFT. In *Proceedings 3rd European Workshop on Advanced Mobile Robots (EUROBOT)*. Zürich, Schweiz, September 1999.

[131] C. Schlegel and R. Wörz. The software framework SMARTSOFT for implementing sensorimotor systems. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1610–1616. Kyongju, Korea, October 1999.

[132] D. Schmidt. ACE - Adaptive Communication Environment.
URL *http://www.cs.wustl.edu/˜schmidt/ACE.html*.

[133] D. Schmidt. TAO - Realtime CORBA with TAO.
URL *http://www.cs.wustl.edu/˜schmidt/TAO.html*.

[134] D. Schmidt and S. Vinoski. Object Interconnections - An Overview of the OMG CORBA Messaging Quality of Service (QoS) Framework. *C++ Report, SIGS*, 12(3), March 2000.

[135] D. C. Schmidt. Monitor object – an object behavior pattern for concurrent programming, (updated october 10th). *C++ Report, SIGS*, 12(4), May 2000.

[136] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 1*. C++ In-Depth Series. Addison-Wesley, 2002.

[137] D. C. Schmidt and S. D. Huston. *C++ Network Programming, Volume 2, Systematic Reuse with ACE and Frameworks*. C++ In-Depth Series. Addison-Wesley, 2003.

[138] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2*. John Wiley and Sons, Ltd., 2000.

[139] R. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, 1994.

[140] R. Simmons. The curvature-velocity method for local obstacle avoidance. In *Proc. IEEE International Conference on Robotics and Automation*, pages 3375–3382. Minneapolis, April 1996.

[141] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Victoria, Canada, 1998.

[142] R. Simmons and Nak Young Ko. The lane-curvature method for local obstacle avoidance. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Victoria B.C., Canada, 1998.

[143] R. C. Smith, M. Self, and P. Cheeseman. Estimating uncertain spatial relationships in robotics. In I. J. Cox and G. T. Wilfong, editors, *Autonomous Robot Vehicles*, pages 167–193. Springer-Verlag, 1990.

[144] W3C Web Services Activity: SOAP (Simple Object Access Protocol).
URL *http://www.w3.org/TR/SOAP/*.

[145] C. Stachniss and W. Burgard. An integrated approach to goal-directed obstacle avoidance under dynamic constraints for dynamic environments. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2002.

[146] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, Harlow, England, 1998.

[147] S. Thrun, M. Bennewitz, W. Burgard, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. MINERVA: A second generation mobile tour-guide robot. In *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 1999.

[148] S. Thrun, W. Burgard, and D. Fox. A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning*, 31(5):1–25, 1998.

[149] J. Uhlmann. *Dynamic Map Building and Localization: New Theoretical Foundations*. PhD thesis, Robotics Research Group, Department of Engineering Science, University of Oxford, 1995.

[150] I. Ulrich and J. Borenstein. VFH+: Reliable obstacle avoidance for fast mobile robots. In *Proc. IEEE International Conference on Robotics and Automation*. Leuven, Belgium, 1998.

[151] I. Ulrich and J. Borenstein. VFH*: Local obstacle avoidance with look-ahead verification. In *Proc. IEEE International Conference on Robotics and Automation*. San Francisco, 2000.

[152] H. Utz, S. Sablatnög, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, 18(4):493–497, 2002.

[153] E. Waltz and J. Llinas. *Multisensor Data Fusion*. Artech House, 1990.

[154] W3C Architecure domain: Extensible Markup Language (XML).
URL *http://www.w3.org/XML/*.

Zusammenfassung

Servicesysteme in Form von mobilen Robotern sollen zukünftig Aufgaben in einer natürlichen Umgebung ausführen, teilweise sogar zusammen mit Personen. Derartige Systeme müssen ihre Umgebung wahrnehmen und auf wahrgenommene Veränderungen selbständig reagieren. Das Ziel ist oftmals nicht mehr maximale Effizienz, sondern die zuverlässige und vor allem selbständige Bewältigung verschiedener Aufgaben.

Nach heutigem Stand des Wissens verlangt dies eine integrierte Nutzung symbolischer und subsymbolischer Mechanismen der Informationsverarbeitung mit vielfältigen Wechselwirkungen. Während subsymbolische Mechanismen Flexibilität und Reaktivität auch in veränderlichen Umgebungen sicherstellen, erlauben symbolische Ansätze zielgerichtete Handlungsabläufe, erfordern aber meist sehr viel Zeit und andere limitierte Ressourcen in kritischen Ablaufbereichen.

Die *state-of-the-art* Architektur für solche Systeme ist eine 3-Ebenen-Architektur. Die unterste Ebene, genannt *skill layer*, besteht aus Komponenten, die Sensorwerte verarbeiten und Aktoren ansteuern. Diese bilden Regelungsschleifen, erstellen eine Karte der Umgebung oder stellen Kollisionsfreiheit bei Bewegungen sicher. Die mittlere Ebene, genannt *sequencing layer*, ist für die situationsabhängige Selektion und Konfiguration von Komponenten zuständig. Sie koordiniert die Aufgabenausführung durch Synchronisieren des Ausführungsfortschrittes mit der diskreten Beschreibung der erforderlichen Aktionsfolge. Die oberste Ebene, *deliberation layer* genannt, umfaßt zeitintensive Algorithmen wie die symbolische Handlungsplanung, die typischerweise zukünftige Zustände berücksichtigen.

Die organisatorische Stärke einer 3-Ebenen-Architektur ist in ihrer grundsätzlichen Unterstützung einer situationsabhängigen Anwendung der auf dem Roboter verfügbaren Fähigkeiten begründet. Dies ist vor allem für Servicesysteme wichtig, die eine Vielzahl unterschiedlichster Herausforderungen während der Aufgabenausführung bewältigen müssen. Typischerweise reicht ein einzelner Ansatz für jede Teilaufgabe nicht aus, da es unwahrscheinlich ist, daß beispielsweise eine Bewegungsführung alle Aufgaben der Bewegungsführung bewältigen kann.

Eine 3-Ebenen-Architektur entfaltet ihre volle Leistungsfähigkeit nur auf der Basis einer ausreichenden Anzahl verfügbarer Basisfähigkeiten. Weiter müssen diese dafür ausgelegt sein, zu unterschiedlichen Verhaltensmustern kombiniert zu werden. Ohne die Möglichkeit der situations- und aufgabenabhängigen Konfiguration und Rekombination von Basisfähigkeiten ist man typischerweise nicht einmal in der Lage, selbst eine einfache Hol- und Bringaufgabe auszuführen. Weiterhin ist man nicht in der Lage, Schwachstellen einzelner Ansätze durch gezielte Auswahl von Alternativen auszugleichen.

Sobald eine steigende Anzahl von Komponenten zur Ausführung unterschiedlichster Aufgaben und zur Bewältigung verschiedenster Situationen benötigt wird, ergeben sich sofort zwei Aspekte. Zum einen sind auf einem mobilen System nur beschränkte Ressourcen verfügbar, so daß Basisfähigkeiten benötigt werden, welche ausgeglichen sind in Bezug auf erforderliche Ressourcen, Reaktivität und erzieltem Resultat. Zum anderen stellt sich sofort die Frage der Komplexitätsbeherrschung. Die

Komplexität ergibt sich nicht nur aus den Einzelkomponenten, sondern vor allem durch ihr wechselndes Zusammenspiel. Obwohl sich 3-Ebenen-Architekturen als Standard herauskristallisiert haben, ist das Fehlen von Standards für Komponenten und deren Interaktion die hauptsächliche Ursache für die Komplexitätsprobleme bei der Umsetzung einer Roboterarchitektur.

Daher besitzt die vorliegende Arbeit zwei Schwerpunkte. Der erste Schwerpunkt zielt auf die Vereinfachung der Realisierung komplexer Roboterarchitekturen wie sie für Servicesysteme benötigt werden. Der zweite Schwerpunkt zielt auf die Erweiterung der Basisfähigkeiten durch ausgeglichene Ansätze, welche die Vorteile lose gekoppelter Komponenten nutzen.

Zur Komplexitätsbeherrschung wird ein Komponentenansatz vorgestellt, der an den Erfordernissen von Mehrebenenarchitekturen ausgerichtet ist. Der Ansatz macht keine Vorschriften bezüglich der Aufteilung in Komponenten, erzwingt aber die Einhaltung wohldefinierter Komponentenschnittstellen. Dies wird erreicht durch standardisierte Kommunikationsmuster, welche eine vordefinierte Semantik für die Interaktion von Komponenten bereitstellen. Dabei kann das *dynamic wiring*-Pattern als *das* Interaktionsmuster der Robotik gesehen werden, da es sowohl den Kontrollfluß als auch den Datenfluß von außerhalb einer Komponente konfigurierbar macht.

Weiter wird ein Ansatz zur Navigation in dynamischen Umgebungen erläutert, der ausgeglichen ist in Bezug auf benötigte Rechenleistung, erzielte Reaktivität und Abweichung von Optimalität und Vollständigkeit. Auf der Ebene der Bewegungsführung wird die exakte Kontur des Roboters berücksichtigt. Die Zuständigkeiten zwischen den einzelnen Komponenten sind so verteilt, daß sich der Roboter immer in einem sicheren Bewegungszustand unter Berücksichtigung der kinematischen, dynamischen und geometrischen Beschränkungen befindet.

Außerdem wird ein Berechnungsverfahren für die simultane Lokalisierung und Kartierung beschrieben. Das Verfahren erlaubt die Akquise einer vorläufigen Karte der Umgebung, welche bereits gleichzeitig zur Lokalisierung und Navigation verwendet werden kann. Durch das Erstellen einer relativen Karte können zeitintensive Optimierungsverfahren zur Bestimmung der besten Karte ohne Informationsverlust aufgeschoben werden. Durch Anwendung eines konservativen Filters können dennoch für die Exploration geeignete Positionsschätzungen einschließlich einer Varianzangabe bestimmt werden.

Die Tragfähigkeit der Ansätze wird anhand eines vollständig realisierten mobilen Roboters illustriert, der unterschiedliche und umfangreiche Hol- und Bringaufgaben in einer dynamischen Testumgebung ausführen kann. Es bestätigte sich, daß sowohl das Gesamtkonzept zur Integration als auch die Methoden, welche die Basisfähigkeiten realisieren, geeignet sind, komplexe Aufgaben unter den schwierigen Bedingungen einer realen Welt auszuführen.

Der Komponentenansatz wurde erfolgreich innerhalb verschiedener Großprojekte sowohl aus dem akademischen als auch dem industriellen Umfeld eingesetzt und bestätigte seine Eignung als Mittel zur Komplexitätsbeherrschung. Zwischenzeitlich stellt dieser Ansatz eine zentrale Komponente der *OROCOS* Initiative dar.