# Communication Patterns as Key Towards Component Interoperability

Christian Schlegel[1]

University of Applied Sciences Ulm
Fakultät Informatik
Prittwitzstr. 10, D-89075 Ulm, Germany
schlegel@hs-ulm.de

## 1 Introduction

A component based software engineering approach does not per se ensure that independently developed components finally fit together. The reason is that general purpose component based approaches still provide far too much freedom with respect to defining and implementing component interfaces. There is always a gap from a general purpose software component approach towards reusable and easily composable components in a specific application domain.

The challenge of component based software approaches for robotic systems is to address the numerous functional and non-functional requirements of this application domain. As argued in Chapter *Trends in Component-Based Robotics*, the major characteristics are related to the *inherent complexity* of robotics systems, the *requirements for flexibility* and the challenges of *distributed environments*.

Of particular importance in the robotics domain are the following aspects that need to be addressed explicitly when tailoring the general idea of component based software engineering to the needs of the robotics domain:

- Components at different levels of a robot system can follow completely different designs. Thus, roboticists ask for as few restrictions as possible with regard to component internal structures.
- *Dynamic wiring* of components can be considered as *the* pattern in robotics. It is the basis for making both, the control flow and the data flow configurable at run-time. That is the basis for situated and task dependent composition of skills to behaviors.

Furthermore, a framework is accepted by roboticists if it provides an obvious surplus value, requires a flat learning curve only, is a software framework and not a robotic architecture, addresses middleware and synchronization issues and provides lots of device drivers and components for widespread sensors and platforms.

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 183–210, 2007.

## 2 The Approach

The basic idea to address the above requirements is to master component interfaces since these are crucial with respect to component interoperability. Mastering the component interfaces also allows to abstract from middleware aspects and to ensure decoupling to address the complexity issue. That idea is implemented by means of a small set of generic and predefined communication patterns. All component interactions are squeezed into those predefined patterns. Thus, component interfaces are only composed out of the same set of well-known patterns with a precise and predefined semantics. This ensures decoupling of components, enforces the appropriate level of abstraction at the externally visible component interfaces and thereby results in components that are composable to form complex robotics applications.

### 2.1 Communication Patterns to Define Component Interfaces

**Components:** A component can contain several threads and interacts with other components via predefined communication patterns that seamlessly overcome process and computer boundaries. Components can be dynamically wired 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, provides predefined access methods and hides all the communication and synchronization issues. It always consists of two complementary parts named *service requestor* and *service provider* representing a *client/server*, *master/slave* or *publisher/subscriber* relationship.

**Communication Objects** parameterize 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 communication when accessing an attribute. Furthermore, object responsibilties are much simpler to manage with locally maintained objects than with remote objects (see [HV99] for nifty details of *CORBA* lifecycle issues). Communication objects are ordinary objects decorated with additional member functions for use by the framework. Genericity of the approach is achieved by using arbitrary and individual communication objects to instantiate communication patterns.

**Service:** Each instantiation of a communication pattern provides a service. Generic communication patterns become services by binding the templates by types of 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. Major characteristics are as follows.

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

### 2.2 The Set of Communication Patterns

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. 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. Table 1 shows the set of communication patterns. A service provider can handle any number of clients concurrently.

Communication patterns make several communication modes explicit like a *oneway* or a *request/response* interaction. Push services are provided by the *push newest* and the *push timed* pattern. Whereas the *push newest* pattern can be used to irregularly distribute data to subscribed clients whenever updates are available, the latter distributes updates on a regularly basis. The *event* pattern is used for asynchronous notification if an event condition becomes true under the activation parameters and the *wiring* pattern covers dynamic wiring of components at runtime.

**Table 1.** The set of communication patterns

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

### 2.3 Reference Implementation

The *CORBA* based reference implementation called SmartSoft uses the *TAO ORB* [Schb] and can be found at [Sma05]. The operating system abstractions are provided by the *ACE* package [Scha][SH02][SH03] so that interoperability and usage across most operating systems is ensured.

Using communication patterns as means to master the otherwise inevitably exploding diversity of component interfaces has been proposed first in [SW99b]. The approach matured over several projects and a most complete description is contained in [Sch04]. Further aspects are described in [Sch06] and [SW99a].

## 3 The User View

The user view comprises the application builder view and the component builder view. The main focus is on guiding the component builder without enforcing a particular robot architecture and thus to allow the application builder to compose off-the-shelf components due to replacement independence.

### 3.1 Examples of Deployed Robots

Figure 1 shows some robots based on components with communication patterns as component interfaces. A small part of the overall components is shown in figure 2. All services are instantiations of the small set of generic communication patterns. Components are wired dynamically at runtime according to the executed task and can thus be rearranged dynamically.

The interaction with higher layers like task sequencing for a discrete description of execution sequences is coordinated via events. These not only report successfully reached intermediate goals, for example, but also non-default events like *got stuck*, *no path* and others. This allows the task sequencing layer to decide on the next configuration and to monitor and coordinate the task execution progress.

**Fig. 1.** Different robots composed out of standardized navigation components.

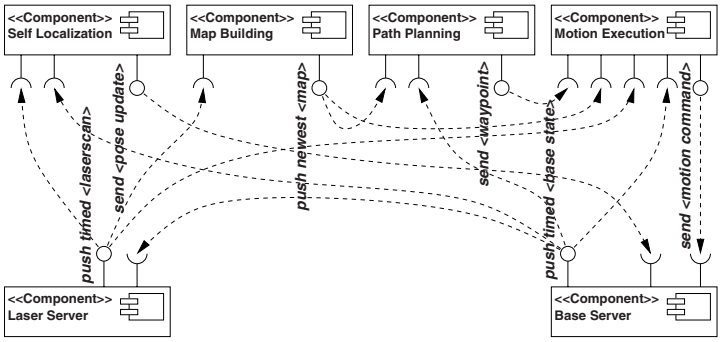


**Fig. 2.** Composability of components due to standardized component interfaces.

### 3.2 User View on the Query Pattern

The user visible interface of the communication patterns is illustrated by means of the *query* pattern. Figure 3 shows the client side user interface that provides both synchronous and asynchronous access modes that can be used in any order and even interleaved without requiring any further access coordination. This also includes the proper handling of blocking calls in case of dynamically performed rewirings of the client to another server.

The server side user interface is shown in figure 4. All incoming requests are forwarded to a handler and answers are returned via the *answer* method. The used handler class defines the processing model like thread per request, thread pool or single thread, for example.

### 3.3 Code Example

The basic usage of the communication patterns to build a component is shown in figure 5. The first component uses a service of the second component implemented by the *query* pattern transmitting a laser scan object. Of course, components can provide and require any number of services and can be interweaved arbitrarily. In this example, the *laserQueryClient* of component *first* is exposed as port *laserPort*. The connections can then even be rearranged from outside the component.

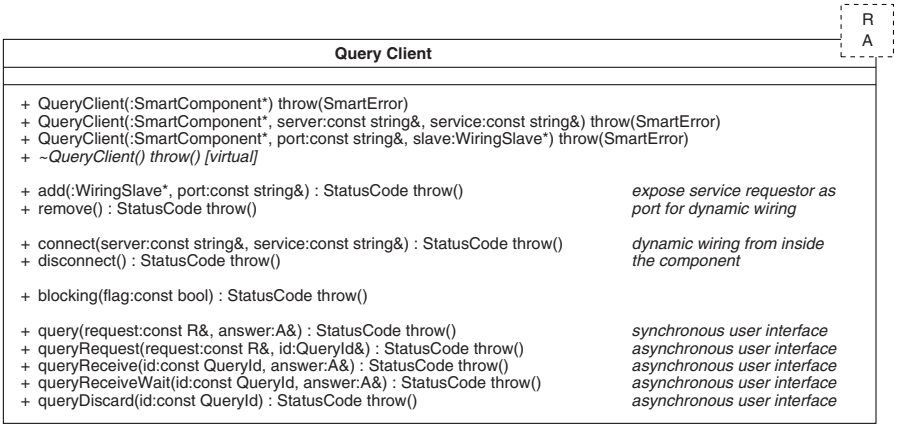


**Fig. 3.** The *query* pattern and its client side user interface.

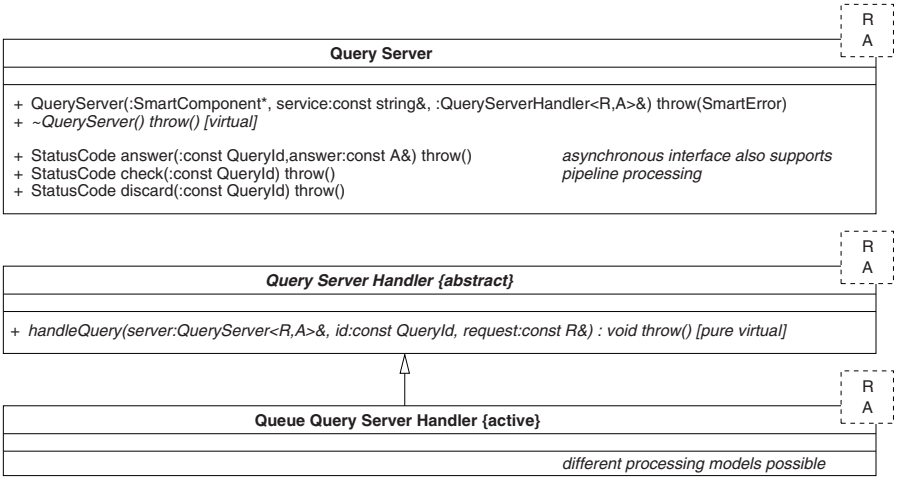


**Fig. 4.** The *query* pattern and its server side user interface.

### 3.4 Communication Objects

Communication objects are transfered *by value*. The relevant data structures are (de)marshalled via the *set/get* methods that form the framework interface together with the *identifier* method. Those methods are the only user visible ones that depend on the underlying communication mechanism. Since these methods can be overloaded, one can implement communication objects such that they work with different middleware systems. The structure of the *CORBA* based implementation is shown in figure 6.

A major advantage is that all user access methods are independent of the middleware data types. In particular, one can use *STL* classes [MDS01] or heap

```
#include "smartSoft.hh"                    // standard include
#include "commVoid.hh"                     // a communication object
#include "commMobileLaserScan.hh"          // a communication object

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

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

int UserThread::svc(void) {
    CommVoid request1, request2;
    CommMobileLaserScan answer1, answer2;
    CHS::QueryId id1, id2;
    ...
    status = laserQueryClient->connect("second","laser");
    status = laserQueryClient->queryRequest(request1,id1);
    status = laserQueryClient->queryRequest(request2,id2);
    ...
    status = laserQueryClient->queryReceiveWait(id2,answer2);
    status = laserQueryClient->queryReceiveWait(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"                    // standard include
#include "commVoid.hh"                     // a communication object
#include "commMobileLaserScan.hh"          // a communication object

// 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::QueueQueryHandler<CommVoid,CommMobileLaserScan>
                        activeLaserHandler(laserHandler);
    CHS::QueryServer<CommVoid,CommMobileLaserScan>
                        laserServant(component,"laser",activeLaserHandler);
    ...
    // the following call operates the framework by the main thread
    component.run();
}
```

**Fig. 5.** Two example components named "first" and "second".

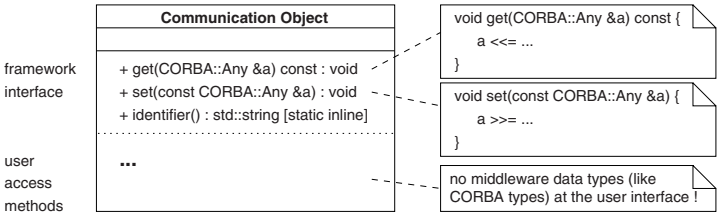


**Fig. 6.** Structure of a communication object.

memory even if these cannot be represented via the *CORBA IDL*, for example. Since all conversions are handled inside the *set/get* methods, one can easily flatten *STL* lists by iterating through them, for example. Furthermore, local extensions of the user access methods have no influence on the transmitted content such that other components relying on the same set of communication objects are not affected. Introducing additional user access methods has purely local effects.

## 4 The Framework Builder View

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 system. These details are neither seen by the application builder nor the component builder.

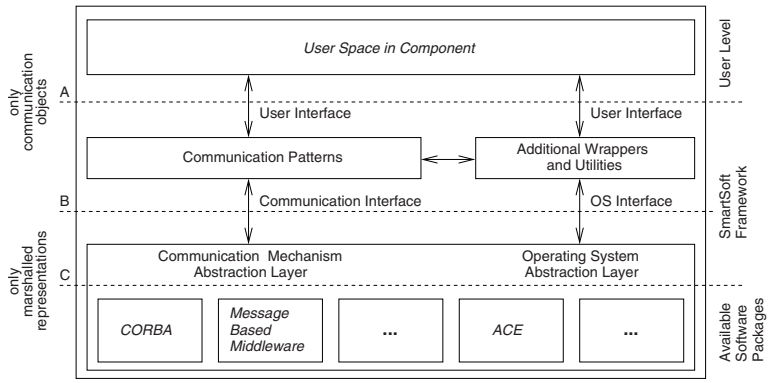


**Fig. 7.** The communication patterns provide the glue logic between the user interface and the underlying communication mechanism.

Communication patterns provide the glue logic which coordinates everything on top of a communication mechanism to enforce the required behavior of the user interface. As shown in figure 7, communication patterns provide the glue logic between the user interface and the underlying communication mechanism. 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 labeled with *A*.

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 nontrivial task since not all communication mechanisms provide all the required features. For example, implementing an asynchronous user interface on top of a synchronous communication mechanism requires carefully chosen protocols. 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.

### 4.1 Interaction Patterns

Communication patterns provide different user access modes like a synchronous invocation, an asynchronous invocation or a handler based processing of incoming requests or answers. From the user level perspective, user access modes provided at the service requestor and those provided at the service provider of a communication pattern can be used in any combination.

From the framework builder view, all reasonable combinations of a user access mode at service requestors and service providers form an *interaction pattern*. The semantics of the user interfaces of the communication patterns and the user access modes assignable to the communication patterns are independent of the underlying communication system as soon as all interaction patterns are implementable on top of the selected communication system.

Interaction patterns 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 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.

#### The Client Part Characteristics

The characteristics of the client part of the interaction patterns are shown in figure 8. 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. They are also 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

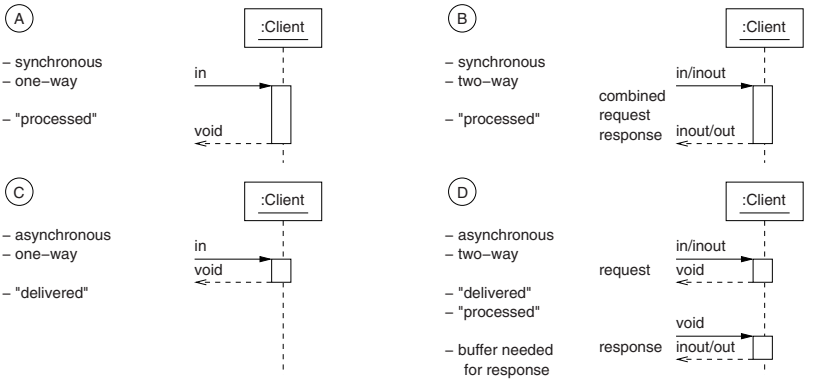


**Fig. 8.** Overview on the characteristics of the client part of the interaction patterns.

arguments can be transmitted solely from the client to the server. In two-way mode, 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 completed 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 might 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 Server Part Characteristics

The server part characteristics of the interaction patterns are shown in figure 9. 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. 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

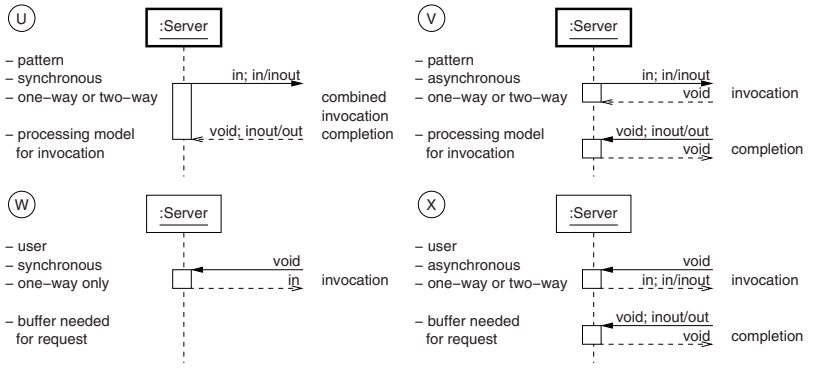


**Fig. 9.** Overview on the characteristics of the server part of the interaction patterns.

to be called from the user level to provide the *out* parts of the modified *in-out* 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. 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. However, pipeline processing models with a synchronous upcall block the upcalling thread until the pipeline returns the answer.

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.

### The Remaining Interaction Patterns

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, only six different interaction patterns remain on top of which all reasonable combinations of user access modes of the communication patterns can be implemented. Figure 10 shows the remaining interaction patterns.

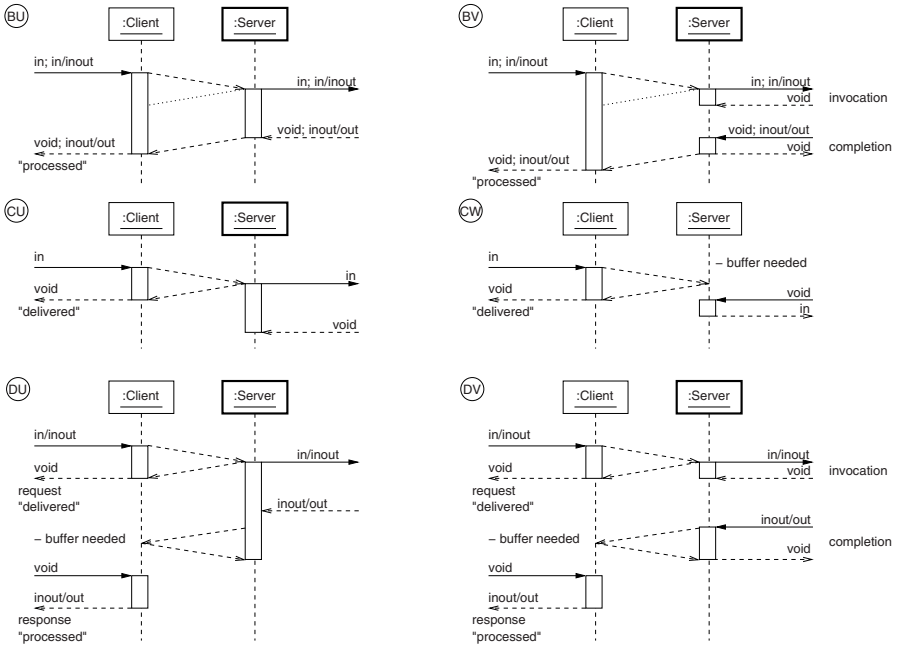


**Fig. 10.** The remaining interaction patterns.

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 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.

### Mapping Interaction Patterns onto Interaction Models

Various communication and middleware systems provide different interaction models. For example, the standard CORBA model provides synchronous remote method calls (B/U), oneway messages (C/U) or asynchrononous method invocations (AMI, D/U). In contrast thereto, a mailbox system or TCP sockets behave like a (C/W) interaction pattern.

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. The idea is to emulate all interaction patterns by solely using the (C/U) interaction pattern. Then 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.

The main advantage is the lean interface between the communication patterns and the communication system as shown in figure 11. *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

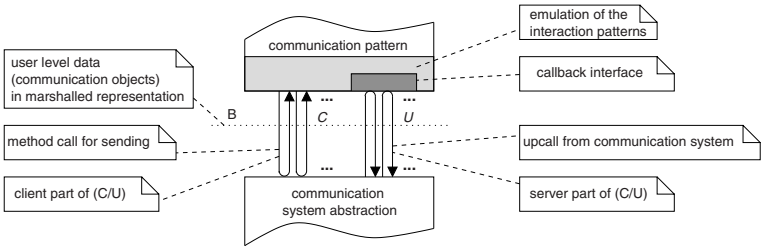


**Fig. 11.** The interface between the communication patterns and the underlying communication mechanism.

are invoked by the upcalling server part of a (C/U) interaction pattern. Of course, this 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 communication patterns and not with every single 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 pattern thus pays off very soon.

It is important to note that there are two features of the (C/U) interaction pattern that predestinate it as interface pattern. Due to its *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 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.

The naive usage of synchronous interactions to emulate the (C/U) interaction pattern can make asynchronous user interfaces obsolete as shown in figure 12. The request returns only after the server side processing has been completed and even after the server has already returned the results. 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.

In case of a *reliable send* policy messages without a regular recipient get dropped silently by the communication system. A recipient can disappear due to destruction, for example. If one cannot be sure that the recipient still exists, one has to use timeouts when awaiting an acknowledgment message. An

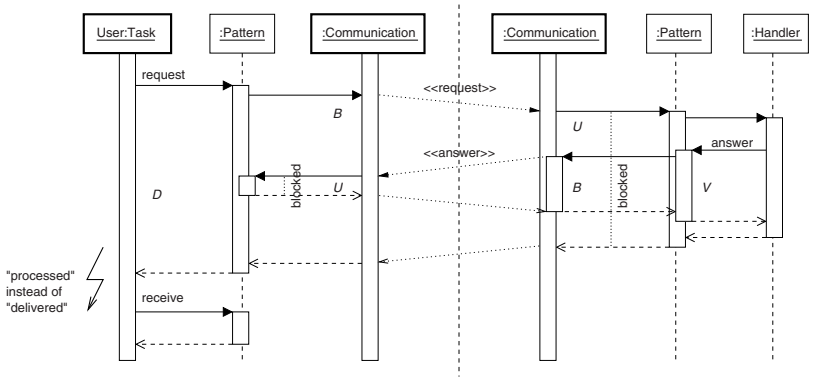


**Fig. 12.** A passive handler and a (D/V) interaction pattern emulated by two (B/U) interaction patterns.

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 could get out of sync which is in contrast to the *delivered* policy of the (C/U) interaction pattern. Thus, one cannot simply map the (C/U) interaction pattern onto an interaction model with a *reliable send* policy only. On top of a *reliable send* policy, one has to achieve the guarantee of the *delivered* policy that is requests get processed for sure once the sending activity got completed successfully.

### 4.2 The Communication Protocol of the Communication Patterns

Another 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 pattern onto the weaker *reliable send* policy without requiring further glue logic at the interface to the communication system. However, that is possible only if the interaction of both parts of a communication pattern are protected by an appropriate protocol as it is the case for the *connection oriented* protocol presented subsequently.

The *connection oriented* protocol that is used inside the communication patterns also requires one-way interactions only and possesses the same lean interface to the communication system as the before introduced protocol. 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 a communication pattern. Changes to a connection can be made at any time and the connection management assumes the responsibil-

ity 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* 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* protocol can even be mapped onto a *reliable send* 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 to be 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* protocol since the decoupling has to take effect for the service related interactions only and can thus be done by and hidden in the user level processing models of the handlers. Standardized handler classes providing active queues, thread pools or other processing models already due to the job.

To summarize, administrative interactions can be mapped onto a *delivered*, a *processed* and a *reliable send* policy without requiring additional mechanisms. Service related data messages on top of a *processed* policy become decoupled by provided user level handler models, in any case work on top of a *delivered* policy and now also work with a *reliable send* policy.

### The (C*/U) Interactions of the Communication Patterns

The interactions between both parts of a communication pattern are emulated on top of the (C*/U) interaction pattern. Figure 13 illustrates the (C*/U) interaction patterns 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.

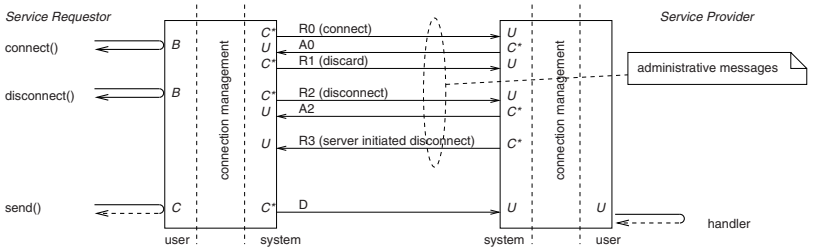


**Fig. 13.** The (C*/U) interactions of the *send* communication pattern.

The messages R0, R1 and R2 can be emitted by all service requestor parts of communication patterns, the acknowledgment messages A0 and A2 and the message R3 by all service provider parts.

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 an underlying *reliable send* policy. A *connect* is rejected if the service provider is either not yet ready or is already in the process of destruction. In depth explanations of the connect/disconnect procedures and how they cope with a *reliable send* policy, service provider destructions and concurrent service provider destructions and connects of service requestors are contained in [Sch04].

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 *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 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 14. 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.

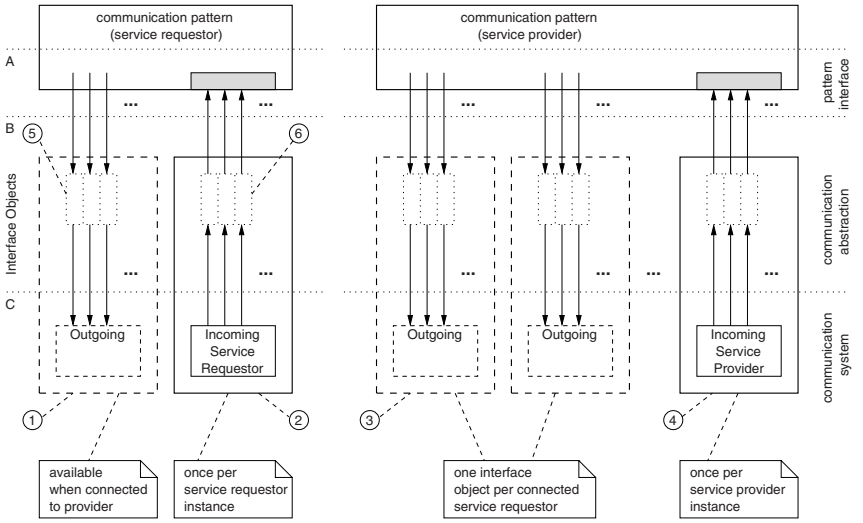


**Fig. 14.** The generic structure of the connection between the communication patterns and the communication system.

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 de-

structed as soon as the corresponding service requestor gets disconnected. The interface object ④ gets destructed with the shutdown of the service provider.

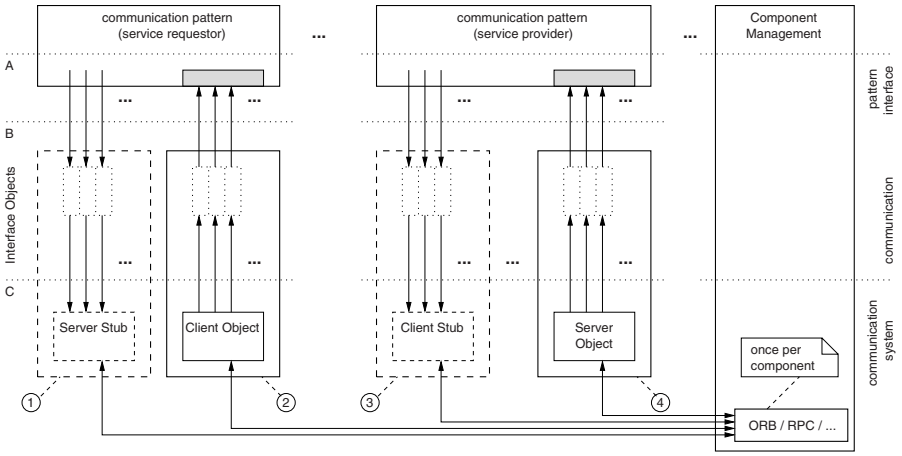


**Fig. 15.** The interface objects of the communication patterns with an object based middleware.

Figure 15 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. This kind of interface is completely independent of the capabilities of the actual *CORBA ORB* since one does neither depend on *oneway-messages* [SV00] [OSK00] nor on *value-types* or *asynchronous method invocations (AMI)* [AOS00].

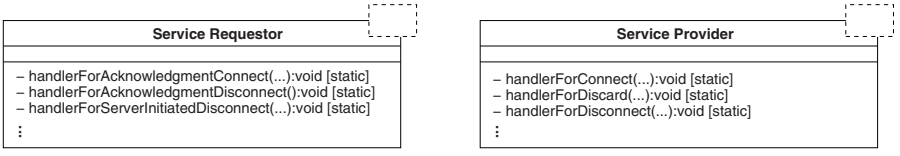


**Fig. 16.** Implementation details of the upcall interface of the communication patterns.

Figure 16 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.

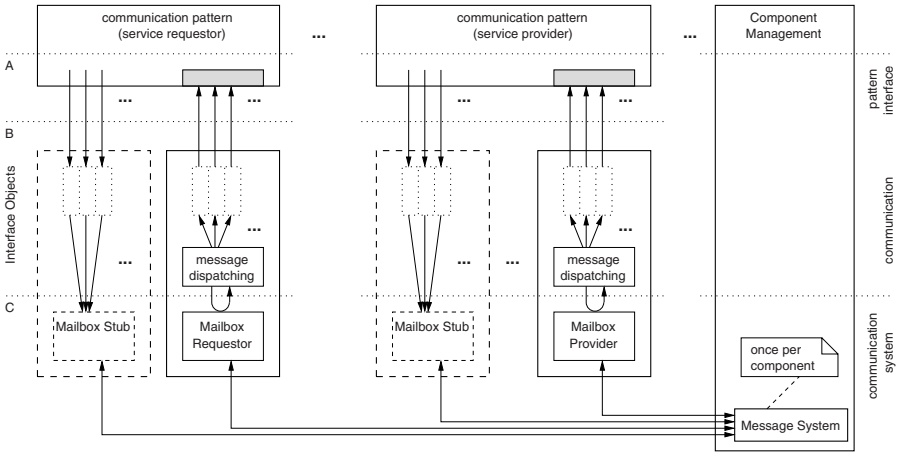


**Fig. 17.** The interface objects of the communication patterns with a message based middleware.

Figure 17 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 [SSRB00] 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 Internal Locking Strategy 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* 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 overall locking strategy is explained by means of the administrative (B/U) interaction. 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.

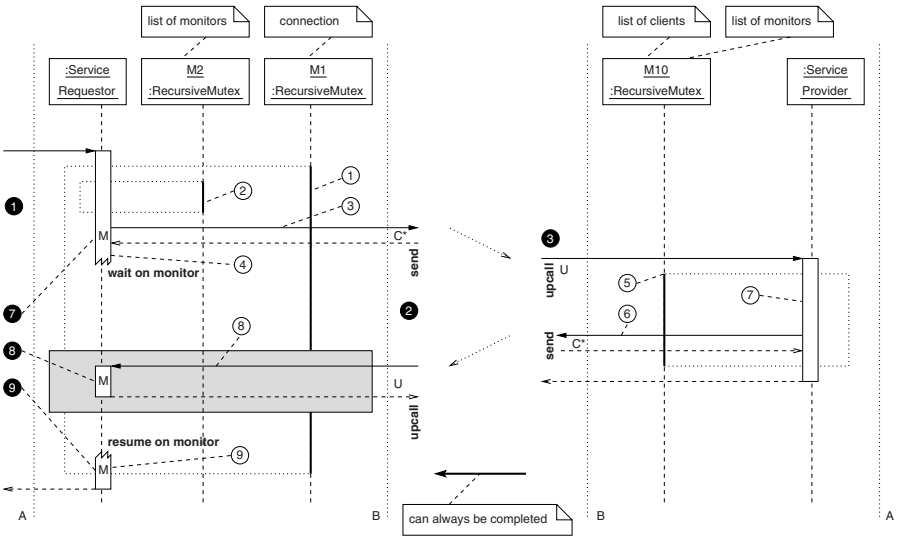


**Fig. 18.** Administrative (B/U) interaction from the service requestor to the service provider (*delivered* policy).

Figure 18 illustrates an administrative (B/U) interaction from the service requestor to the service 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. It sets the state automaton to *await*. The *M* in the upcall ❽ sets the state automaton inside the monitor according to the received answer and performs a broadcast. 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 *sub-*

*scribed* 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* [ED94]. 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 18 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 and answered 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* 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.

Figure 19 shows the same administrative (B/U) interaction on top of a *processed* policy. It is important to note that even the nested callback due to the *processed* policy cannot result in a deadlock.

### 4.3 Details of 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 requires to clean up not yet fully processed requests at the server. All patterns are based on the same mechanisms illustrated by the *query* pattern, that is monitors and state automatons to handle the activities of concurrent requests.

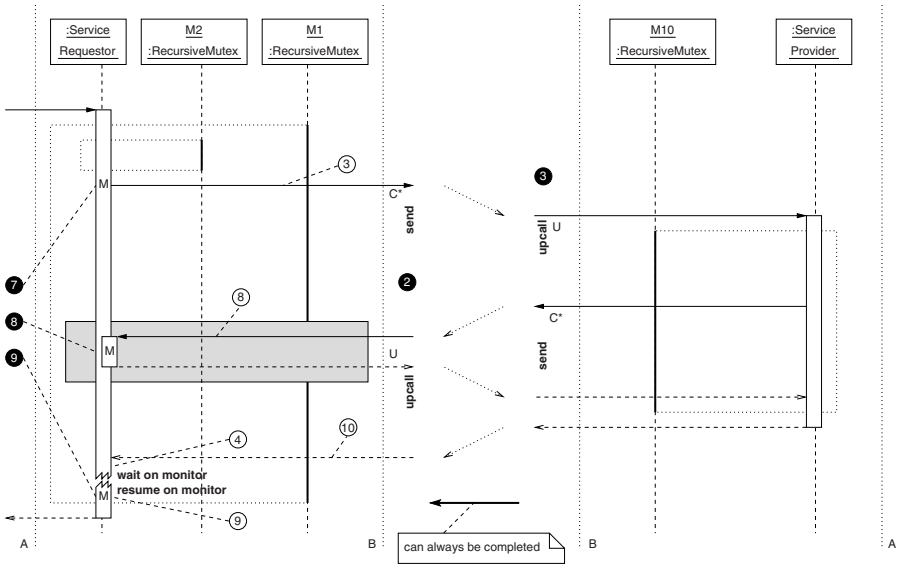


**Fig. 19.** Administrative (B/U) interaction from the service requestor to the service provider (*processed* policy).

## The Client Part

All active queries are represented by their own monitor instance that wraps a state automaton, contains a unique query identifier and provides storage for the received answer until it is retrieved. The client side state automaton is shown in figure 20. The query identifiers manage the basic version of an asynchronous completion token [SSRB00]. Those automatons are the basis for the correct processing of pending calls in case of dynamic rearrangenments of service interconnections by the dynamic wiring pattern.

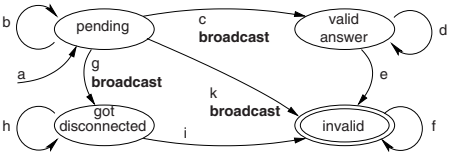


**Fig. 20.** The client side state automaton to manage the lifecycle of a query.

The sequence diagram of the client side upcall to process an incoming answer is shown in figure 21 and the internals of the client side *receiveWait* method are shown in figure 22.

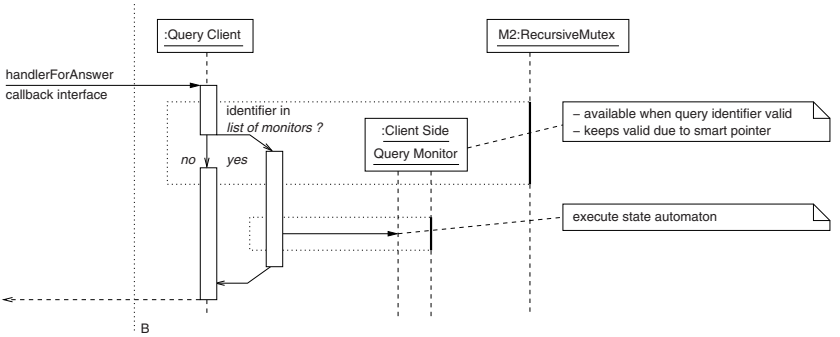


**Fig. 21.** The internals of the client side handler for the *answer* message.

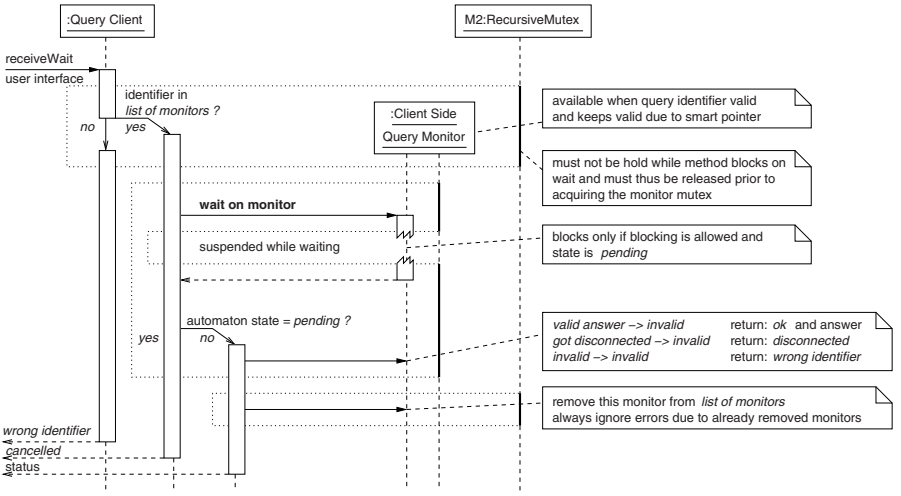


**Fig. 22.** The internals of the client side *receive wait* method.

## 5 Summary and Conclusion

Communication patterns are more than just hiding middleware complexity from the robotics expert. They avoid dubious interface behaviors and do not restrict the component internal architecture. The underlying middleware mechanism is fully transparent and can even be exchanged without affecting the component interfaces. Implementations with the very same features and behavior are available on top of *CORBA*, *TCP* sockets, Mailboxes, *RPC* based communication and message based systems. Standardized communication objects for maps, laser range scans and other entities further simplify the interoperability of components. Dynamic wiring is the key towards dynamic and task and context dependent composition of control and data flows.

The approach of communication patterns closes the gap between the general idea of component based software engineering and composability by re-

ducing the diversity of externally visible component interfaces and by prescribing the interface semantics.

## References

[AOS00] 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*, Middleware, 2000, pp. 208–230.

[ED94] J. R. Ellis and D. L. Detlefs, *Efficient Garbage Collection for C++*, Usenix Proceedings, February 1994.

[HV99] M. Henning and S. Vinoski, *Advanced CORBA programming with C++*, Addison-Wesley, 1999.

[MDS01] D. R. Musser, G. J. Derge, and A. Saini, *The STL tutorial and reference guide, second edition*, Addison-Wesley, 2001.

[OSK00] 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*, Sixth IEEE Real Time Technology and Applications Symposium (RTAS), May 2000.

[Scha] D. C. Schmidt, *ACE - Adaptive Communication Environment*, http://www.cs.wustl.edu/∼schmidt/ACE.html.

[Schb] D. C. Schmidt, *TAO - Realtime CORBA with TAO*, http://www.cs.wustl.edu/∼schmidt/TAO.html.

[Sch04] C. Schlegel, *Navigation and execution for mobile robots in dynamic environments - an integrated approach*, Ph.D. thesis, Faculty of Computer Science, University of Ulm, 2004, http://www.rz.fh-ulm.de/∼cschlege.

[Sch06] C. Schlegel, *Communication patterns as key towards component-based robotics*, ARS Special Issue on Software Development and Integration in Robotics, Int. Journal of Advanced Robotic Systems (2006).

[SH02] D. C. Schmidt and S. D. Huston, *C++ network programming, volume 1*, C++ In-Depth Series, Addison-Wesley, 2002.

[SH03] 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.

[Sma05] SmartSoft, *Reference implementation*, 2005, http://www.rz.fh-ulm.de/∼cschlege or http://smart-robotics-sourceforge.net/.

[SSRB00] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture, volume 2*, John Wiley and Sons, Ltd., 2000.

[SV00] D. C. Schmidt and S. Vinoski, *Object Interconnections - An Overview of the OMG CORBA Messaging Quality of Service (QoS) Framework*, C++ Report, SIGS **12** (2000), no. 3.

[SW99a] C. Schlegel and R. Wörz, *Interfacing different layers of a multilayer architecture for sensorimotor systems using the object-oriented framework* SMARTSOFT, Proceedings 3rd European Workshop on Advanced Mobile Robots (EUROBOT) (Zürich, Schweiz), September 1999.

[SW99b] C. Schlegel and R. Wörz, *The software framework* SMARTSOFT *for implementing sensorimotor systems*, Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (Kyongju, Korea), October 1999, pp. 1610–1616.