A Component Approach for Robotics Software: Communication Patterns in the OROCOS Context

Christian Schlegel

Research Institute for Applied Knowledge Processing (FAW)
Helmholtzstr. 16, D-89081 Ulm, Germany
schlegel@faw.uni-ulm.de

Abstract. Vital functions of robots are provided by software and software dominance is still growing. Mastering the software complexity is not only a demanding but also indispensable task towards an operational robot. Component based software approaches provide suitable means to master the complexity issue. Nevertheless shareable, distributable and reusable off-the-shelf software components for robotics are still a great dream. One of the reasons is the lack of a software component model taking into account robotics needs. The challenge of component based software approaches for robotic systems is to assist in building a system and to provide a software architecture without enforcing a particular robot architecture.

This paper presents communication primitives as core of a robotics component model. Dynamic wiring of components at run-time is explicitly supported by a separate pattern which tightly interacts with the communication primitives. This makes the major difference to other approaches. Advantages provided are software reuse, improved maintainability and software reconfiguration on-the-fly. The presented approach already proved its fitness in several major projects. The *CORBA* based implementation is freely available and is maintained and continued as part of the open source project *OROCOS* [1,9].

1 Introduction

For a long time *integration* has been considered to require only a minor effort once the needed algorithms are all available. The difficulties to overcome have been vastly underestimated.

An important step from laboratory prototypes towards everyday robots is increased reliability and robustness. This requires to master the inherent complexity of robotic systems. Component based approaches address the complexity issue by splitting a complex system into several independent units with well-formed interfaces. Fitting of components is ensured by standards for their external appearance and behavior. This allows to compose systems of approved components and to focus on a single component when going into details without bothering with internals of other components.

A component based approach is not only useful for robotics hardware but is also advantageous at the software level. This is in particular true with regard to the still growing dominance of software in robotics. So far, there is hardly a chance to share software components between labs or reuse them on another platform even for the most often needed skills like localization or motion control. The lack of standard specifications for robotics software requires error-prone and tedious reimplementations wasting valuable resources.

The presented patterns are not yet another framework but already proved their fitness in several major projects [7][13]. The approach has first been presented in [10] and formerly formed the SMARTSOFT framework. The current *CORBA* based implementation [8] is maintained as part of the *OROCOS* [1] open source component framework for robotics. This ensures both continuity in development and maintenance and broad notice in the robotics community. Both is besides easy usage and apparent added value for component developers crucial for establishing a software framework. A growing number of contributed components can pave the way towards robotics applications assembled out of standard software components.

2 Requirements

According to the *OROCOS* project several categories of users are distinguished which all put a different focus on complexity management for integration in robotics.

End users operate an application based on the provided user interface. They focus on the functionality of their application and use a readily provided system with a given functionality to fulfill the required tasks. They do not care on how the application has been built by the application builder 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.

A component based software approach per se already tackles many of the above demands. The following compact definition developed at a workshop is altogether still appropriate:

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.“ [14]

The main difference to object oriented approaches 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, components explicitly consider reusable pieces of software that have well specified public interfaces, can be used in unpredictable combinations and are stand-alone entities. Important features required in the robotics domain which go beyond standard component based software are the following ones:

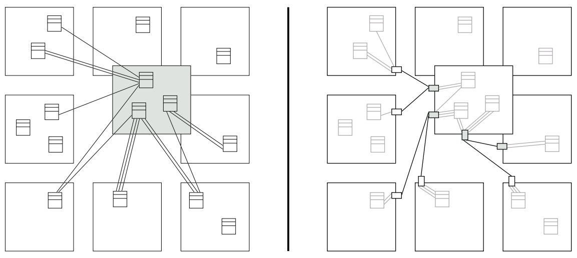


Fig. 1. Component interweaving with fine-grained component interfaces versus the proposed more abstract service based interfaces.

Dynamic wiring can be considered as *the* pattern of robotics. It allows changes to connections between services of components to be made at runtime. Making both the *control flow* and the *data flow* configurable from outside a component is for example the key to situated skill compositions and is required in nearly any robotics architecture. The dynamic wiring pattern tightly interacts with the communication primitives and makes the major difference to other approaches. Reconfigurable components are modular components with the highest degree of modularity. Most important, they are designed to have replacement independence.

Component interfaces have to be defined at a reasonable level of granularity to restrict spheres of influence and to support loosely coupled components. As shown in figure 1, too fine-grained component interfaces can still 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 shows puzzle-like replacement of components where internals are fully decoupled from the externally visible interfaces.

Asynchronicity is a powerful concept to decouple activities and to exploit concurrency as far as possible. Decoupling is in particular important at the component level to avoid passing on tight timing dependencies between components. A robotics framework has to exploit asynchronicity whereever possible without involving the framework user and should make the use of asynchronous interactions as simple as using synchronous ones.

Component internal structures can follow completely different designs depending on the used algorithms. Component builders thus ask for as less restrictions as possible but expect the framework to ensure interoperability by assisting in structuring and implementing a component.

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 since full transparency often not only results in a decreased performance but also prevents predictability.

Easy usage allows focusing on robotics and avoids steep learning curves by making up-to-date software technology available without requiring a robotics expert to become a software engineering expert. 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.

3 Related Work

An often neglected aspect of available or proposed frameworks is that a sensorimotor system is composed of many and heterogeneous algorithms. The requirements in terms of modularity, configurability, communication and control are critical and have to be considered altogether. This explains the need for a framework with domain specific patterns. General purpose component architectures can of course significantly simplify the implementation of the framework and can provide the underlying infrastructure.

Frameworks are so far mainly provided by the robot manufacturers and are in most cases vendor specific. For example, *Mobility* [4] is an up-to-date and *CORBA* [2] based package available with the platforms of ISI. Although this is already much more than is provided by many other vendors, it still only provides an object centered view and therefore is mainly useful for simplified access to the robot's hardware. Other systems like *Saphira* [5] already implement a specific architecture.

GenoM [6] as another framework in the context of *OROCOS* focuses on a specific component internal architecture which is independent of the used component communication mechanism. For example, its requirements on a component communication mechanism are fully matched by the presented approach.

CORBA [2] is a vendor-independent standard for distributed objects that is being extended continuously with the number of supported features growing rapidly and now also comprises an advanced and elaborate component model [3]. In the beginning, lack of features made *CORBA* unsuitable for robotic applications. For example, asynchronous interfaces and *object by value* semantics made their way into *CORBA* very late. Now the countless options and services show a remarkable complexity and thus make *CORBA* a tool for framework builders who select the best fitting mechanisms and hide the underlying middleware from the application builder and from the component developer.

The *CORBA* based implementation [8] of the proposed communication patterns uses *TAO* [12] as ORB and *ACE* [11] as operating system abstraction layer. It takes advantage of many and already standardized features like the host independent *IDL*. However, the communication patterns can be put on top of many other middleware systems like message based systems, remote procedure calls or sockets without requiring much efforts since the requirements on the underlying communication system are very low.

4 The Approach

Mastering the intercomponent communication is considered as the key to master component dependencies and to ensure uniform component interfaces. The developed approach therefore selects intercomponent communication as a suitable starting point.

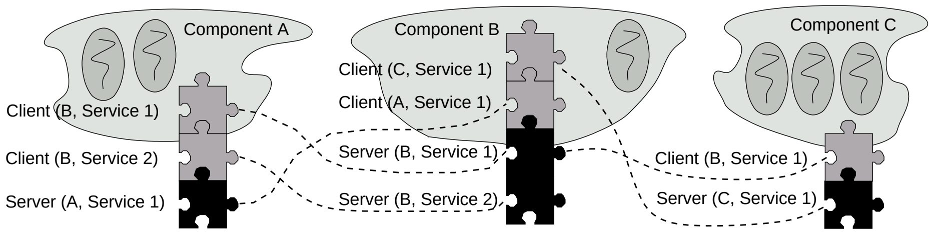


Fig. 2. Overview on the approach.

The basic idea is to provide a small set of communication patterns which can transmit objects between components and then squeeze every component interaction into those predefined patterns. As shown in figure 2, components interact solely via those patterns.

Components are technically implemented as processes. A component can contain several threads and interacts with other components via predefined communication patterns. 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, 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.

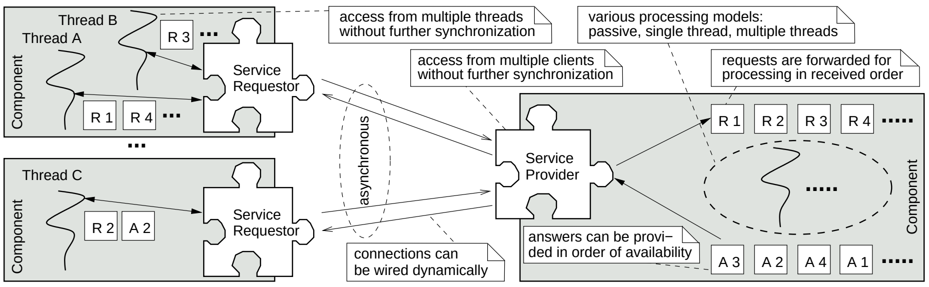
Communication Objects parameterize the communication pattern templates. They represent the content to be transmitted via a communication pattern. They are always transmitted *by value* to avoid fine grained intercomponent communication when accessing an attribute. Furthermore, object responsibilities 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 content as defined by the communication objects.

The set of communication patterns is summarized in table 1. Component interfaces are only composed of services based on standard communication patterns. The communication patterns make several communication modes explicit like an *oneway* or a *request/response* interaction. Push services are provided by the *push newest* and the *push timed* pattern. Whereas the *push newest* can be used to irregularly distribute data to subscribed clients whenever updates are available, the latter triggers calculation and distribution of 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. The set of communication patterns is not the smallest possible one since an one-way communication is already suffi-

Table 1. The set of communication patterns.

Pattern	Relationship	Initiative	Service Provider	Communication
send	client/server	client	server	one-way communication
query	client/server	client	server	two-way request/response
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



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

Fig. 3. 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. The example shows the *query* pattern.

cient to implement any other communication mode. However, it is a reasonable trade-off between usability and minimality. Further hints on the details of the patterns and on why these are sufficient can be found at [8].

Predefined member functions of the patterns provide access modes like synchronous and asynchronous service invocations or provide a handler based request handling. Independently of the access modes and the underlying middleware, the communication patterns always interact asynchronously and thus communication patterns ensure decoupling of the interacting components irrespective of the access modes used by the user. The access modes provide the opportunity to fully handle issues of concurrency, synchronization and decoupling inside the communication patterns hidden from the user instead of dealing with them again and again in every single user defined object visible at a component interface. Communication patterns hide the underlying middleware and do not expect the framework user to for example deal with *CORBA* details like *AMI* and *valuetypes*. Compared to distributed objects, one can neither expose arbitrary member functions as component interface nor can one dilute the precise interface semantics. Both avoids puzzling over the semantics of component interfaces. Individual access methods are moved from the externally visible component interface to communication objects. Since communication objects are always transmitted *by value* and since member functions of communication objects are not exposed outside a component, usage of communication objects and implementing user member functions is completely

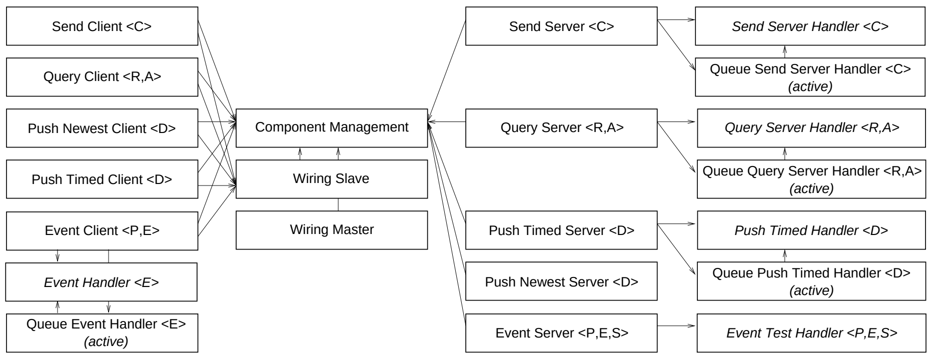


Fig. 4. The core patterns of the SMARTSOFT framework.

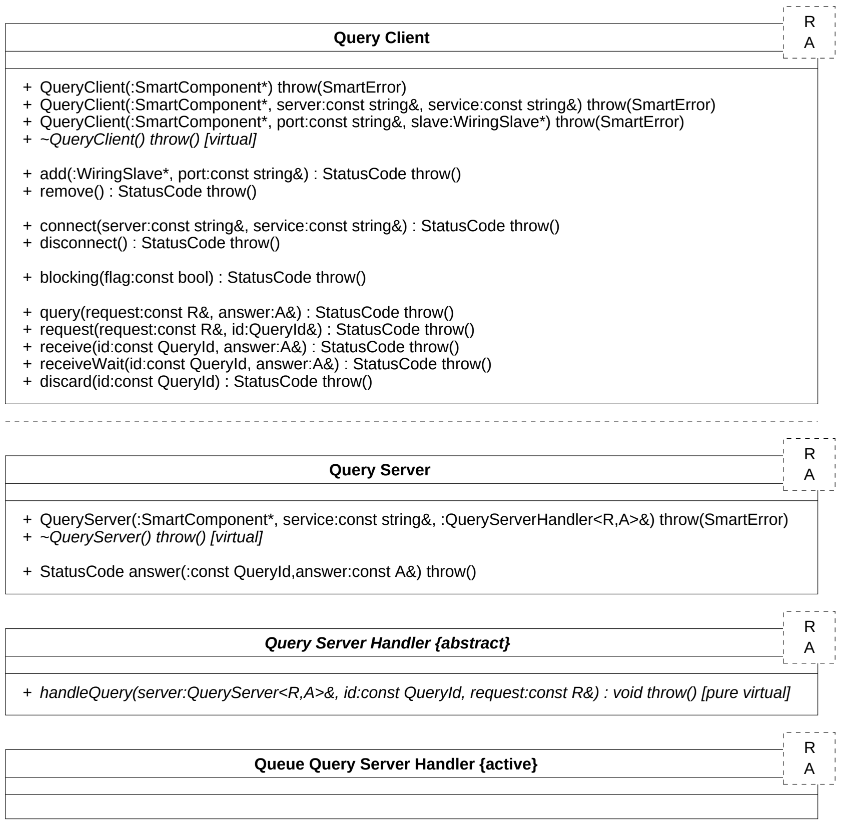
free from cumbersome and demanding details of intercomponent communication and distributed object mechanisms. Arbitrary communication objects provide diversity and ensure genericity even with a very small set of communication patterns. Figure 3 illustrates the key concept and figure 4 summarizes the core patterns of the framework.

5 The Component Builder View

The component builder view is illustrated with respect to the *query* pattern. Figure 5 shows the user API of both the service providing (server) and the service requesting (client) part. Since a *query* requires a *request* and an *answer* object, the pattern template has two communication object parameters. The service requestor always provides several constructors including immediate wiring with a service provider and exposing the client as port wireable from outside. Connections can always be changed using the *connect/disconnect* methods. A client can decide on being wireable from outside the component using the *add/remove* methods. Furthermore, each service requestor provides a *blocking* method to set an internal state indicating whether blocking calls are allowed. If the blocking mode is set to *false*, already blocking calls are aborted and new calls return immediately. These constructors and member functions are part of all service requestors listed in table 1 except the wiring pattern.

The *query* service requestor provides synchronous (*query*) and asynchronous (*request*, *receive*, *receiveWait*) access modes. The *query* service provider expects a handler that is called each time a request is received. The answer is returned by a separate *answer* method to provide an asynchronous server side interface to simplify user level processing models like object driven processing chains. The asynchronous server side interface allows to use active handlers and gives full control over the processing models and the used resources. In contrast, many middleware systems are either single threaded resulting in bottlenecks or provide a thread per invocation where the number of threads often reaches the system limits.

Pending queries are managed by monitors and condition variables. A blocking wait on a pending query does neither waste system resources nor does it block in a member



```
+ 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()
```

```
+ QueryServer(:SmartComponent*, service:const string&, :QueryServerHandler<R,A>&) throw(SmartError)
+ ~QueryServer() throw() [virtual]

+ StatusCode answer(:const QueryId,answer:const A&) throw()
```

```
+ handleQuery(server:QueryServer<R,A>&, id:const QueryId, request:const R&) : void throw() [pure virtual]
```

Fig. 5. User API of the *query* communication pattern.

function of a remote object. The client side management of blocking calls decouples blocking access modes from the service provider and supports a client side canceling of blocking member function calls. For example, this feature is needed by a component state management to rush through blocking calls to reach a forced state as fast as possible and still in an ordered way. The access modes of the other communication patterns are comparably easy to use.

5.1 Dynamic Wiring

As shown in figure 6, the *wiring* pattern makes service requestors visible as ports and wireable at runtime from outside the component. All service requestors listed in table 1 except the wiring pattern itself can expose themself as wireable ports. Wiring of services is based on names. Services are denoted by $\{$*component name, service name*$\}$ and ports by $\{$*component name, port name*$\}$. The wiring master provides a *connect* method to connect a port with a service and a *disconnect* method to suspend the connection of

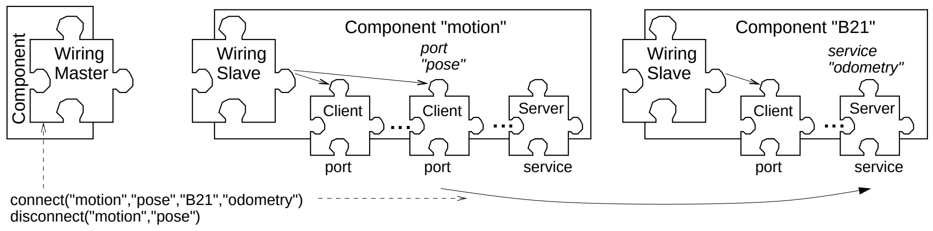
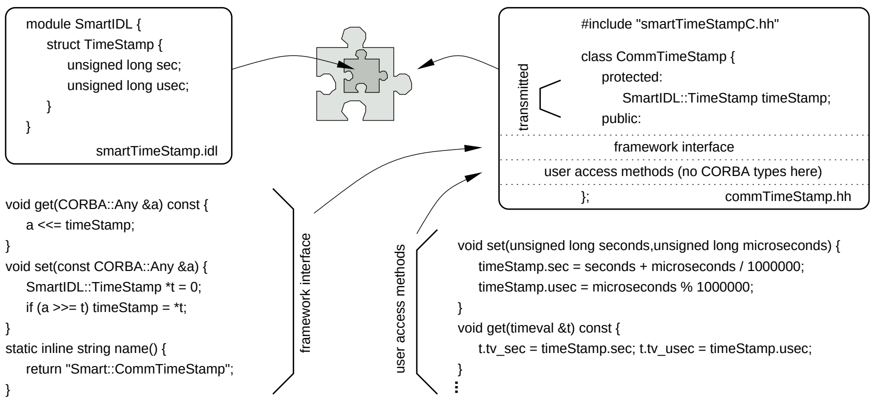


Fig. 6. Making service requestors visible as ports and wireable from outside the component.



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

```
#include "smartTimeStampC.hh"

class CommTimeStamp {
    protected:
        SmartIDL::TimeStamp timeStamp;
    public:
.................................................
            framework interface
.................................................
    user access methods (no CORBA types here)
.................................................
};                               commTimeStamp.hh
```

```
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";
}
```

```
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;
}
...
```

Fig. 7. Implementation of the communication object for a time stamp as example.

a port. A service requestor and a service provider are compatible if both the communication pattern and the communication object types match.

6 Example

The following example illustrates the simple usage of the communication patterns. As component builder, one normally first checks the availability of standardized and reusable communication objects. Otherwise, one has to define a new communication object type. This requires to describe the data structure to be transmitted in *CORBA IDL* and to add the appropriate additional member functions for use by the framework as shown in figure 7. This is the only place where one gets in touch with the underlying middleware. When using predefined communication objects, one does not even see those details. Normally, there is already a huge set of agreed communication objects available which can be reused and which ensure interoperability of components. It is important to note that no middleware specific *CORBA* types are exposed at the user interface of the communication objects. One can for example use well-known *STL* classes irrespective of the *IDL* described data structure used for transmission inside the communication object.

```
#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::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. 8. Two example components named „first“ and „second“.

The example in figure 8 consists of the components *first* and *second* which require respectively provide laser scans based on the *query* pattern. A *void* object without further parameters is used to request a laser scan object. The service requestor is wireable as *laserPort*, the service provider uses a handler to process requests.

7 Conclusion

To develop component based software, it is necessary to draw clean boundaries between each component. Many attempts of creating component based software fail because designers have no guidelines to follow when breaking up the application into pieces. We define a subsystem as part of an application that can be developed and tested independently and integrated into an application later through simple communication mechanisms. There are no precise rules for decomposition but support for clean arrangements of component interfaces based on standardized communication objects and services. Components based on the presented approach can more easily be used in multiple configurations.

Using communication patterns as core of a component approach ensures clearly structured component interfaces and avoids dubious interface behaviors while still not restricting the component internal architecture. Moving access modes from the user domain into the communication patterns ensures decoupling of components by only using asynchronous interactions at the intercomponent level inside the patterns. Communication objects prevent the component from being poluted with middleware data types. Standard communication objects for maps, laser range scans, ultrasonic sensor values etc. support uniform representations at the component level. Using inheritance one can individually extend a communication object inside a component with individual member functions without affecting other components. Applying the communication patterns does not require additional software expertise beyond the standard knowledge. Due to its simple usage, it might boost the component based development of robotics software.

The communication protocol used inside the communication patterns can be used on top of many different communication systems and does not at all depend on *CORBA*. The decoupling is even ensured on communication systems that only provide synchronous interactions.

The presented approach has already been used successfully in several major projects where it proved its fitness. The set of already available components comprises standard robot platforms, sensors, algorithms and visualization and is growing rapidly. An open source implementation is available within the OROCOS project.

References

1. H. Bruyninckx. Open robot control software: The OROCOS project. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 2523–2528. Seoul, Korea, May 2001.
2. CORBA, Object Management Group, Inc. (OMG). *http://www.corba.org/*.
3. CORBA Component Model, Object Management Group, Inc. (OMG). *http://www.omg.org/*.
4. iRobot. *Mobility 1.1 Robot Integration Software User's Guide*, 1999.
5. K. Konolige. *Saphira Robot Control Architecture Saphira Version 8.1.0*. SRI International, April 2002.
6. 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 Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 2292–2297. Lausanne, Switzerland, October 2002.
7. MORPHA: Interaction, communication and cooperation between humans and intelligent robot assistants. *http://www.morpha.de/*.
8. FAW contributions to the OROCOS project. *http://www1.faw.uni-ulm.de/orocos/*.
9. The OROCOS project. *http://www.orocos.org/*.
10. C. Schlegel and R. Wörz. The software framework SMARTSOFT for implementing sensorimotor systems. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 1610–1616. Kyongju, Korea, October 1999.
11. D. Schmidt. ACE - Adaptive Communication Environment. *http://www.cs.wustl.edu/˜schmidt/ACE.html*.
12. D. Schmidt. TAO - Realtime CORBA with TAO. *http://www.cs.wustl.edu/˜schmidt/TAO.html*.
13. SFB 527: Integration of symbolic and subsymbolic information processing in adaptive sensorimotor systems.
14. C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, Harlow, England, 1998.