

Communication Patterns as Key Towards Component-Based Robotics

Christian Schlegel

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

Abstract: Vital functions of mobile 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. Nevertheless, well-known and even always needed algorithms are often implemented from scratch over and over again instead of being reused as off-the-shelf components. A major reason is the lack of a framework that allows to compose robotics software out of standardized components without already prescribing a robotics architecture. Whereas Component Based Software Engineering (CBSE) is an approach to make a shift from implementation to composition, the general CBSE approach does not give any hints on how to ensure that independently provided components finally fit together as reusable components. This paper introduces a small set of communication patterns as basis for all intercomponent interactions. Since all externally visible interfaces of components are composed out of the same set of communication patterns, these are the key towards a stringent interface semantics. Generic communication patterns enforce decoupling of components and ensure composability by restricting the diversity of interfaces. The SmartSoft framework as one implementation of this approach already proved its adequacy in many projects.

Keywords: software engineering, component approach, software reuse, communication pattern, robotics

1. Problem Statement

One of the reasons for the lack of off-the-shelf robotics software components 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. In particular in robotics, there are many demanding functional and non-functional requirements.



Fig. 1. Different robot platforms operated by SmartSoft reusing many components for navigation skills.

From the technical point of view, robotics software always requires to cope with the inherent complexity of concurrent activities, the deployment of software components on networked computers ranging from embedded systems to personal computers, a bunch of platforms, operating systems and programming languages, the requirement to hide distribution aspects by a middleware mechanism, timing and resource constraints and of course also with organizational

challenges of distributed development processes and issues of integration of independently developed components (fig. 1).

From the point of view of a component builder, the focus is on specifying and implementing a single component. A framework is expected to provide the infrastructure that supports the implementation effort in such a way that the component finally is compatible with other components without being restricted too much in respect of component internals. A component builder wants to focus on algorithms and component functionality without bothering with integration issues.

From the point of view of an application builder, applications should be composable out of off-the-shelf, standardized and thus reusable components. An application builder expects a framework to ensure clearly structured and consistent component interfaces for easy assembling of approved components.

Although component based software approaches already proved that they provide the right level of granularity to address the above requirements, there is still a gap from a general purpose software component approach towards reusable and easily composable components.

This paper argues that component approaches still provide far too much freedom in matters of component interfaces resulting in non-reusable components since externally visible interfaces of components almost never

fit together neither at the level of abstraction or interface methods nor at the level of interface semantics.

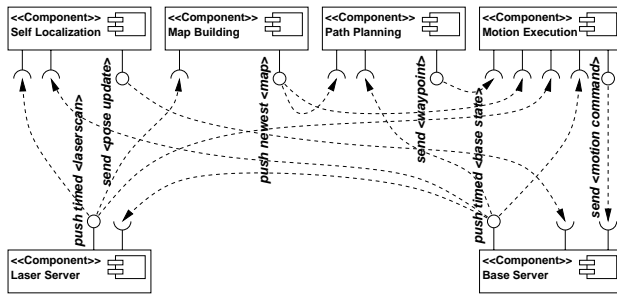


Fig. 2. Communication patterns to master the diversity of component interfaces.

Thus, a small set of generic and predefined communication patterns with a strictly defined interface semantics and an appropriate level of decoupling as the only way to define component interfaces are suggested to overcome this gap. Fig. 2 shows some of the components on board a mobile platform distributed over several networked computers. The wiring of components is dynamic and can be reconfigured online according to the tasks to be executed.

2. Related Work

A component is defined as 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” (Szyperski, 1998). Major existing component models comprise COM+/Microsoft, Enterprise JavaBeans/Sun and the CORBA Component Model (Heineman & Councill, 2001). The first one is proprietary and not portable across operating systems, the second one is available only within JAVA and the last one still lacks reliable and mature implementations. This view is also shared by (Brooks, 2005). Often, component models are quite complex and require substantial programming skills. Thus, robotics experts have to get familiar with advanced software techniques in addition to their main interests. All component models have in common that they do not provide any assistance or hints on how to best define the externally visible component interface. Of course, this often is domain specific and also requires lots of domain specific experience. Indeed, it is exactly this kind of knowledge that has to be made available in such a way that best practices can easily be adopted by newcomers. Then they can contribute to robotics in a consistent way without being discouraged by a software complexity hurdle.

Using communication patterns as means to master the otherwise inevitably exploding diversity of component interfaces has been proposed first by (Schlegel & Wörz, 1999). The approach matured over several projects and a most complete description can be found in (Schlegel,

2004). The reference implementation called SmartSoft can be found at (SmartSoft, 2005). It already proved its adequacy as general software engineering approach even outside robotics (Avitrack, 2004). Meanwhile, the approach of using communication patterns has been adopted by other groups that not only provide another code base but further prove that communication patterns are an appropriate way of introducing CBSE concepts into robotics (Brooks, 2005).

Many different robotics software frameworks have been proposed so far. Most of them are just object oriented approaches without an explicit component model (Montemerlo, 2003, Vaughan, 2003, Utz, 2002), enforce a particular component internal structure (Mallet, 2002) or even prescribe a specific robotics architecture (Konolige, 1997). None of them assist the component builder in defining appropriate intercomponent interaction mechanisms.

A state of the art survey on software engineering in robotics focusing on design decisions behind matured frameworks is given in (Brugali, 2005).

3. Functional and Non-Functional Requirements

Dynamic wiring can be considered as the pattern in robotics. Dynamic wiring allows changes to connections to be made at runtime. It is the basis for making both, the control flow and the data flow configurable. That is the basis for situated and task dependent composition of skills to behaviors.

Component interfaces have to be defined at a reasonable level of granularity. A reasonable level avoids fine-grained intercomponent interactions and supports loosely coupled components but with a stringent and standardized interface semantics.

Asynchronicity: Patterns of a robotics framework should make use of asynchronicity wherever possible. A system consists of loosely coupled components that run asynchronously and communicate with each other. The usage of asynchronous interactions by the component builder should be as simple as possible.

Component internal structures: Components at different levels of a robot system can follow completely different designs. Component builders therefore ask for as few restrictions as possible with regard to component internal structures.

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

Easy usage: Acceptance of a framework for sensorimotor systems is increased by offering obvious additional value while avoiding steep learning curves. Providing access to state-of-the-art software technology without requiring every component builder to be a software engineering

expert allows robotics experts to focus on algorithms and relieves them from the burden of software integration. Challenging topics which have to be addressed are, for example, location transparency of components and their services and concepts of concurrency including synchronization and thread safety while not ignoring different bandwidth requirements (vision systems). A framework provides additional value only if its usage is much simpler than matching all the requirements without using a framework.

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.

Introducing a new framework is a tough business since there is nearly never the right point of time to get rid of the code base grown in a research group over several years. The only possible way is a component repository that is growing by contributions of others and that has the obvious advantage of providing cutting-edge technology components in a composable way. As soon as composability really works, no research group will spend any time anymore on reimplementing something for which others already gained their merits.

4. Communication Patterns as Key To Composability

The basic idea is to provide a small set of generic communication patterns that can transmit objects between components.

A major purpose of communication patterns is to relieve the component builder from error-prone details of distributed and concurrent systems by providing approved and reusable solutions. Communication patterns have to provide patterns for higher level component interactions.

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.

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 responsibilities are much simpler to manage with locally maintained objects than with remote objects. 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.

Service: Each instantiation of a communication pattern provides a service. Generic communication patterns become services by binding the templates by types of communication objects.

send	one-way communication
query	two-way request/response
push newest	1-to-n distribution
push timed	1-to-n distribution
Event	asynchronous conditioned notification
Wiring	dynamic component wiring

Table 1. The set of generic communication patterns.

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

Looking at the external interface of a component immediately opens up their provided and required

services, and looking at the communication pattern underlying a service immediately opens up the usage and semantics of that service.

The implementation of the user accessible member functions of the communication objects are neither affected by aspects of the intercomponent communication nor can these implementations introduce any intercomponent dependencies. In accordance with the service based view, communication objects can be accessed and manipulated without entailing external component interactions. They can also be used independently of the subsequent state of the service providing component once they have been obtained. Arbitrary member functions defined by the user can be implemented without taking into account any aspects of intercomponent dependencies since their execution context never spans across components.

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

5. Technical Details

Both parts of a communication pattern can provide completely different access modalities since both parts are not only forwarding method calls but are standalone entities. For instance, the *query* pattern shown in fig. 3 provides both, synchronous and asynchronous access modalities at the client side and a handler based interface at the server side.

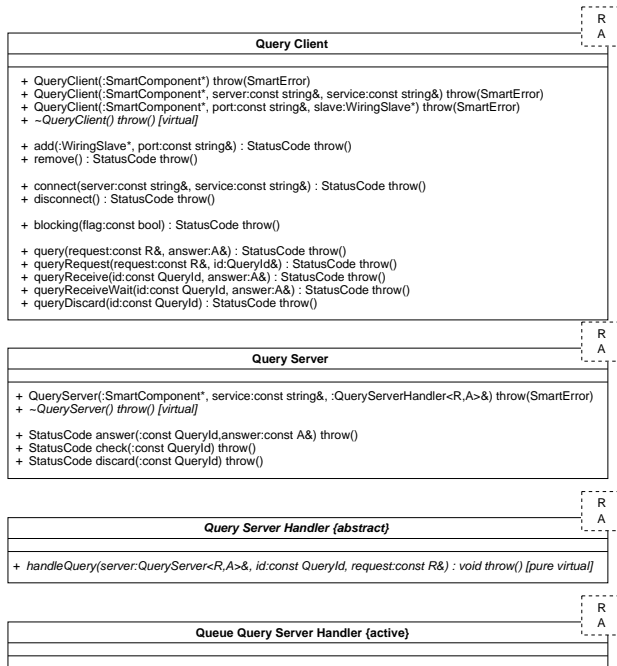


Fig. 3. The *query* pattern and its user interface.

The pattern internal communication is based on oneway messages that can be mapped onto arbitrary

communication systems without affecting the behavior of the user interface.

The communication mechanism can be mapped onto standard middleware systems like CORBA (synchronous, asynchronous method calls) as well as on top of simple message based systems (mailboxes) or even raw TCP/IP connections. This is completely transparent to the user. The message oriented abstraction of the internally used communication protocol in combination with sophisticated synchronisation mechanisms ensures that even with an underlying synchronous communication system (resulting in nested calls from the client to the server and back to the client) no deadlock can occur and that still all asynchronous requests immediately return after *delivering* a request and not after having *processed* it.

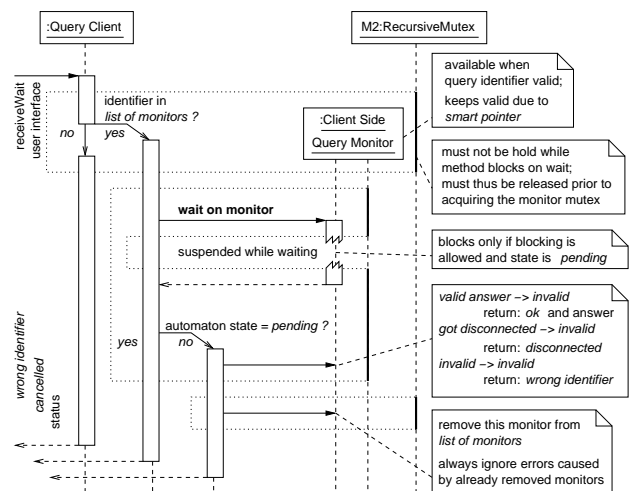


Fig. 4. The *receiveWait* method of the *query* client.

As detailed in fig. 4 by means of the *query* pattern, each request is managed by its own monitor that wraps a state automaton. As shown in fig. 5, the state automaton represents all states of a request including whether it cannot be expected anymore due to a disconnect that occurred meanwhile etc. The various mutexes ensure that independently of the order of user method calls, dynamic wirings and characteristics of the underlying communication system, neither a deadlock can occur nor that an asynchronous interaction gets converted into a synchronous one (see (Schlegel, 2004) for full details).

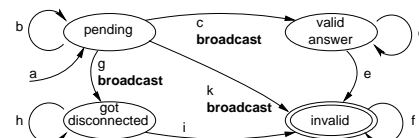


Fig. 5. Client side state automaton of the *query* pattern.

6. Dynamic Wiring

The *wiring* pattern supports dynamic wiring of services from outside (and inside) a component by exposing service requestors of a component as ports. They can be identified non-ambiguously via their names and the

name of the component. The wiring pattern allows to connect service requestors to service providers dynamically at run time. A service requestor is connected only to a compatible service provider. A service provider and a service requestor are compatible if they are instantiated by the same types of communication objects and if they belong to the same type of communication pattern. Disconnecting a service requestor automatically performs all housekeeping activities inside a communication pattern to sort out not yet answered and pending calls, for example, by iterating through the affected state automata and thus properly unblocking method calls that otherwise would never return. For example, the wiring mechanism already properly sorts out effects of a server getting destroyed while clients are in the process of connecting to it (see (Schlegel, 2004) for full details). Of course, this relieves a component builder from a huge source of potential pitfalls.

7. Lessons Learned

This section reviews design decisions made with the approach of communication patterns. It reviews normally hidden aspects that proved to be useful and are stable over all kinds of applications of mobile robots.

Communication patterns result in component interfaces with clear semantics since all component interfaces are always composed out of the same patterns and thus all expose the same methods. This greatly simplifies commitments on component functionality and interfaces and allows testing and replacement with low effort. It also supports agreements on deliverables, interfaces and responsibilities within a project of distributed partners.

Communication patterns are thread-safe and can be accessed in any order in any combination of overlapping requests without requiring any user-level coordination or state maintenance. This relieves the users of the communication patterns from complex and error-prone state management and difficult abortion of blocking calls. Normally, this results in complex component internal structures. Since all these challenges are already handled inside the communication patterns, a great source of errors is already removed from the responsibility of a component builder.

Dynamic wiring is a basic functionality needed in nearly any robotics application to dynamically compose complex functionalities out of distributed components. Too fine-grained component interfaces would make it impossible to rearrange component connections due to unmanageable component internal states.

Dynamic wiring needs to be integrated into the communication patterns since disconnecting and reconnecting requires proper handling of pending answers, for example. When performing a disconnect, all states of the affected interactions are automatically handled inside the communication patterns.

By-value semantics of transferred objects greatly reduces the sphere of influence and local object responsibilities are much simpler to handle than complex object lifetime mechanisms. Furthermore, local object responsibilities greatly reduce network load since, for example, overloaded operators do not unexpectedly expand across component boundaries.

Communication objects can be extended by arbitrary methods without affecting the core data structures that are transmitted via the communication patterns. Thus, additional methods are private to the component and need not to be exposed globally. Locally extending communication objects by inheritance does not affect any other component using the same communication object. Thus, locally needed interfaces can be added locally without affecting already agreed communication objects and without extending globally visible interfaces.

Separating middleware data type descriptions inside the communication objects from the user access methods allows to use any data type independently of the capabilities of the middleware data type description language. For example, one can easily transfer objects that contain heap memory or that use STL classes at user interface methods. Thus, middleware migrations only affect the description inside the communication objects and are not visible outside the communication objects. Furthermore, advances in middleware technology can be seamlessly applied within the communication patterns without affecting the framework semantics. Since no middleware data types are visible outside the communication objects, framework users need not to get familiar with lifetime issues of middleware objects.

The communication patterns can be configured to run transparently on top of different communication systems like shared memory, tcp sockets, middleware systems like CORBA, and others. That is important since robotics applications need to integrate embedded controllers and standard personal computers, sometimes need special links to ensure high bandwidth or have to consider resource constraints. One can even use shared memory based patterns in parallel to others to improve throughput for those components that reside on the same host.

User access methods at both parts of communication patterns (the service provider and the service requestor) are completely independent of each other and all provided user access modes can be used concurrently. Thus, one can use the synchronous and the asynchronous access modes of a service concurrently without further coordination and without implying any processing mode at the other side of the communication pattern. An asynchronous request at the client side can result in a synchronous processing at the server. This flexibility ensures that components can internally implement their individual and most suitable structure without implying restrictions or side effects on internals of other components.

A naming service for components is needed to reduce configuration and deployment efforts as well as basis for the wiring pattern. However, a trader service is likely too difficult in robotics since one has to know too much about the requirements, dependencies, timing of components and other aspects prior to making a proper selection of a service provider. Thus, the trader service is included in the higher levels of a three-layer architecture and is provided as part of the configuration management and task execution mechanisms and not as middleware service.

8. Summary and Conclusion

Communication patterns proved to be a suitable approach towards composability of robotics applications. The approach closes the gap between the general idea of component based software engineering and composability by reducing the diversity of externally visible component interfaces and by prescribing the interface semantics.

Communication patterns provide clean arrangements of component interfaces based on standardized communication objects and services. They enforce loose component couplings by internally using asynchronous component interactions independently of the underlying communication mechanism. Communication patterns 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. This is the prerequisite to implementing robotics architectures based on off-the-shelf and standardized components. Communication patterns also proved their usefulness in order to enforce standardized component interfaces outside robotics applications.

To summarize, the reasons for the success of the communication patterns is foremost based on several small but important design decisions.

Modifications of and further progress at the underlying middleware layer is fully transparent to user since the communication layer beneath the communication patterns is completely separated from the robotics application level. Communication patterns provide a fixed set of visible methods at component interfaces with a clear semantics. User access modes at the client and the server part are completely independent of each other. The internal communication between components is fully asynchronous. No middleware data types are visible at

the user level. Objects are transmitted by value. Communication patterns handle concurrent access and provide location transparency. Dynamic wiring supports the implementation of various robotics architectures out of components. The implementation is based on standards to take advantage of progress achieved in other communities outside the robotics world. The license model is GPL / LGPL to support widespread usage.

9. References

- Avitrack (2004). Task 5.1 Framework Prototype. <http://www.aero-scratch.net/avitrack.html>
- Brooks, A.; Kaupp, T.; Makarenko, A.; Williams, S. & Orebäck, A. (2005). Towards Component-Based Robotics. Proceedings IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), Canada
- Brugali, D. (2005). *Principles and Practice of Software Development in Robotics* (Workshop ICRA 2005), <http://robotics.unibg.it/sdir2005/program.html>
- Heineman, G. T. & Councill, W. T. (2001). *Component-based software engineering: putting the pieces together*. Addison-Wesley
- Konolige, K.; Myers, K. L.; Ruspini, E. H. & Saffiotti, A. (1997). The Saphira architecture: a design for autonomy. *Journal of experimental & theoretical artificial intelligence (JETAI)*, 9(1):215-235
- Mallet, A.; Fleury, S. & Bruyninckx, H. (2002). A specification of generic robotics software components: future evolutions of GenoM on the OROCOS context. Proceedings IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), pp. 2292-2297, Switzerland
- Montemerlo, M.; Roy, N. & Thrun, S. (2003). Perspectives on standardization in mobile robot programming: the Carnegie Mellon navigation toolkit. Proceedings IEEE/RSJ Int. Conference on Intelligent Robots and Systems (IROS), pp. 2436-2441, Las Vegas
- Schlegel, C. & Wörz, R. (1999). The software framework SmartSoft for implementing sensorimotor systems. Proceedings IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), pp. 1610-1616, Korea
- Schlegel, C. (2004). *Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach*. PhD thesis, Faculty of Computer Science, Univ. of Ulm, <http://www.rz.fh-ulm.de/~cschlege>
- SmartSoft. (2005). Reference Implementation. <http://smart-robotics.sourceforge.net/>
- Szyperski, C. (1998). *Component Software – Beyond Object-Oriented Programming*. Addison Wesley, Harlow, England
- Utz, H.; Sablatnög, S.; Enderle, S. & Kraetzschmar, G. (2002). MIRO – Middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation*, 18(4):493-497
- Vaughan, R.; Gerkey, B. & Howard, A. (2003). On device abstractions for portable, reusable robot code. Proceedings IEEE/RSJ IROS, pp. 2121-2127, Las Vegas