On Evaluating Interaction and Communication Schemes for Automation Applications based on Real-Time Distributed Objects

Carlos Eduardo Pereira
DELET/UFRGS
Porto Alegre - Brazil
cpereira@delet.ufrgs.br

Joerg Kaiser
University of Ulm
Ulm - Germany
kaiser@informatik.uni-ulm.de

Carlos Mitidieri Claudio Villela
Informatics Institute/PPGC/UFRGS[1]
Porto Alegre - Brazil
{miti,lbecker,villela}@inf.ufrgs.br

Leandro Buss Becker[1,2]
OvG Universität Magdeburg[2]
Institute for Distributed Systems (IVS)
Magdeburg - Germany

Abstract

This paper compares interaction and communication mechanisms used in distributed control systems, focusing on object-oriented and component-based development. The standard communication model used in distributed object-oriented systems is the remote method invocation. We argue that this client/server oriented model has some severe drawbacks when used in a control system where objects may have to broadcast information, spontaneously communicate environmental changes and where control autonomy is a crucial requirement. Therefore, we compare the traditional way of object invocation with a port-based scheme and the model of event channels. An application scenario from robot control is used to highlight similarities and differences among these mechanisms.

1 Introduction

Smart sensors and actuators, powered by micro-controllers and connected via a communication network support in many ways extensibility, reliability, and cost effectiveness of large control systems. The built-in computational component enables the implementation of a well-defined high level interface that does not just provide raw transducer data, but a pre-processed, application-related set of process variables. The communication network represents on a physical level a standardized interface over which the devices can exchange information. It can be foreseen that the technological advances will allow the integration of such smart devices as a single system-on-a-chip, which may comprise hardware, software and even mechanical components. In essence, each device becomes a configurable building block, which encapsulates data and behavior, can be configured by process specific parameters and communicates process relevant data. Consequently, the interfaces and the functions of these smart components are not just related to the raw physical values of the controlled device but they may include functions related to overall control, supervision, and maintenance issues. In such a system, multiple different sensors will co-operate to augment perception of the environment and actuators will co-ordinate actions to increase speed, power and quality of actuation thus forming virtual sensor and actuator networks. Perhaps the most challenging property of these intelligent devices is their ability to spontaneously interact with the overall system. This enables a modular system architecture in which smart autonomous components co-operate to control a physical process without a central co-ordination facility. This property matches a vital requirement of many real-time systems for modularity and easy configuration to enable incremental system evolution. Particularly because the lifetime of real-time systems is much longer than the fast cycles of technology, it is of utmost importance that components can be replaced without changing the entire system. Additionally, due to the continuous and uninterrupted operation requirement of large real-time systems, this reconfiguration should be made on-line.

The advantages of having such autonomous but co-operative components should be complemented by design methods able to exploit the envisaged distributed structure. Object-oriented design has proven to be one of the key technologies for the development of large and complex systems.

The key concepts of modularization, information hiding, and inheritance gave decisive advantages over other software design methods. The emphasis is on the development process, incremental system extension, and better maintainability of software.

Among the existing methods for developing such distributed, usually embedded, computer-based systems, the adoption of the concept of distributed objects have been frequently mentioned in the literature as particularly adequate, particularly due to their modularization and encapsulation characteristics, leading objects to be relatively self-contained and autonomous. This has an impact on how the interaction between entities has to be organized. The interaction between components comprises communication and co-ordination and is usually related to the overall system design method. In object-oriented systems, a prevailing way of interaction is through remote message invocation. Interaction thus is concerned with higher level issues and should provide an adequate abstraction from the underlying communication network. Since real-time aspects are a key issue when developing such systems, careful attention has to be paid to aspects related to their temporal behavior, which is strongly influenced by the communication patterns adopted.

Another attractive approach to promote flexibility and adaptability in distributed systems is the use of multi-agent systems. Like an object, an agent encapsulates state and methods, but different from it, an intelligent agent has the autonomy to decide at what time it attends to a request or even not to attend, although it could be forced by design to reply to pre-defined query types from particular agents. This introduces new dimensions to all aspects of system development, specially regarding communication. Agents work in a cycle of sensing-knowledge, processing-reasoning, and reaction-actuation. Messages from other agents should be accepted through internally controlled perceptual channels, in order to achieve a predictable timing. Furthermore, to enable the interaction among agents of different capabilities, communication must be defined at several levels, with less capable agents using more restrictive mechanisms. Others factors that influence interaction in multi-agent systems and may rise different needs are legacy software and system architecture. In the first case, a transducer agent could be implemented to translate agent queries into requests to existing programs. Finally, depending on the system architecture, agents can communicate directly or through an inter-mediator, with impact on the binding.

In this paper, three frequently adopted strategies for developing distributed real-time applications based on the concept of distributed objects are compared: remote method invocation [10, 15], port-based communication [3, 8, 14], and publisher-subscriber [1, 2, 6, 12, 9, 7]. Some additional comparisons concerning the content based communication scheme [1, 2, 6, 12, 9, 7] are also found in section 4. The paper aims to discuss the strengths and weaknesses of each approach both in terms of their adequacy for modeling as well as for implementing real-time systems based on distributed objects.

This paper is organized as follows: next section provides a more comprehensive description of each of the above mentioned communication strategies. A case study has been selected for comparing the different approaches. The case study and the obtained results are presented on section three. Section four summarizes a comparison among the different strategies from the viewpoint of their adequacy to the development of distributed real-time systems. Section Five draws the conclusions and outlines directions of future work.

2 Brief overview on selected communication mechanisms

When designing an interaction mechanism for a distributed object-oriented control system where distributed objects are considered as autonomous and concurrent processing units, some important features that should be supported by the underlying communication infra-structure are:

- Many-to-many communication - This is particularly important for control systems composed from intelligent sensors and actuators because the output of a sensor may be used by many entities.
- Spontaneous generation of messages triggered by a timer or by an external event.
- Control autonomy, i.e. co-ordination of activities is orthogonal to communication.
- Independent design and easy extensibility.
- Long term system evolution with incremental compilation possibilities.

In this section, we evaluate and compare different methods of high-level object interaction along this line. At first, the impact of many-to-many communication relations is examined. After that, it is discussed how spontaneous generation of messages is realized in the different methods. This is strongly related to the question of control autonomy. E.g. in a model relying on a request/reply scheme of interaction, communication and co-ordination are generally more strictly coupled as in a mechanism which supports an event driven model. Finally, design issues are discussed, with emphasis on at what time in the design process and how communication relationships (sender/receiver) have to be defined. This can be at design time, configuration time, run time or even can be omitted at all. Clearly, specifying communication relationships at design time makes it difficult to cope with a dynamically changing environment with respect to these relations because of the need to adapt to unanticipated events occurring during the mission of a system. Additionally, long term system evolution usually requires the modification of old components or the creation of new ones. Therefore, it would be desirable to accommodate these changes "on-the-fly" on a running system without disturbing existing components.

2.1 Remote method invocation

In this communication pattern, as usually proposed by object-oriented languages such as C++ and Java, a client object must know (or using programming terminology must have a reference to) a server object. Remote method invocations (RMI) are similar to remote procedure calls [4] in conventional distributed programming, with the difference that in object-oriented languages the called methods exist within the context of a given instance of a class.

This implies that an explicit bind - at design time - must be created between those elements that will interact. A sender object must know his receiver counterpart in order to be able to communicate. Internally, a remote method call is usually translated to a message, containing information related to the receiver object, the method to be invoked, and the method parameters. Message exchange then occurs using the underlying communication infrastructure. In order to have this mapping of method calls to messages as transparent as possible to applications, existing middleware based on distributed objects like CORBA, DCOM and Java RMI introduce communication-specific objects, such as stubs, object adapters or proxies.

The remote method invocation thus characterizes a point-to-point (p2p) communication, in a client/server style, where the client object interface maintains a reference to the server object.

This interaction mechanism can execute either in a synchronous or in an asynchronous fashion. In the first case, a request-reply scheme is adopted, derived from the synchronous remote procedure call. Thereby, when one object invokes a method on another object, its execution flow is blocked and the control is transferred to the invoked method. On the other hand, when the interaction is asynchronous, an unblocking method call occurs, i.e. both sender and receiver objects may continue execution.

One of the advantages of having the communication defined explicitly at design time is that a more rigorous consistency checking regarding the number and type of parameters transferred from sender to receiver can be performed. This can avoid some typical errors of distributed programming using only message passing mechanisms.

2.2 Port-based communication

Port based communication supports a component-oriented development. Objects are interconnected and may communicate by exchanging messages through the so called ports, which are part of the object interface. Related concepts are protocols, which specify the set of valid messages on each port as well as their direction (i.e. incoming and out-coming messages). In this case, the whole behavior of a given class can be specified without any prior knowledge about which objects are going to receive or send messages. According to many authors [3, 14] this may leads to a more modular design process, on which the reuse of pre-defined components is encouraged.

When using ports, an object (also called component [14] or actor [3]) is not restricted to dealing with other objects by way of their interfaces rather than their internals, but is also restricted to dealing with only certain segments of interfaces, as defined by ports. This means that a port exposes a partial set of the receiver methods to the sender, thus allowing an object to present different "faces" to the objects with which it interacts by way of different ports. Some methodologies, like ROOM, have stronger protocol policies and demand that binding between ports can only occur between conjugated ports. Input and output ports are called conjugated when outgoing messages from a port are accepted as incoming messages in the connected port on the another object. In general, the establishment of such a connection occurs at design time prior to system execution.

Considering the component-oriented communication model [14], each component should define precise input and output interfaces. The component interface defines its access points. These points allow clients of a component, usually components themselves, to access the services provided by the component. Since a component can have multiple interfaces, corresponding to different access points, one can consider that a component interface is logically equivalent to a port. Anyway, it should be pointed out that there is no necessity for the use of protocols, once the component will not send messages to specific elements.

2.3 Publisher/Subscriber

This model adopts a content-based producer-consumer oriented style of communication. An object as a producer of a message may spontaneously publish this message. The message can be identified by its content. Other objects interested in this content may subscribe. All objects that have subscribed to a certain content are notified when the respective message is published. Because messages are not routed by address, the communication relation can be determined at run time. The respective local communication subsystems filter the message stream to identify all messages to which local objects have subscribed.

The Publisher/Subscriber (P/S) communication approach reflects an event-based style of object interaction. A publisher spontaneously sends a message that is going to be delivered to all objects which have subscribed. A subscription refers to a message class which is related to the content of the message. The subscriber is then notified, whenever a message of the respective class is sent by the publisher. Therefore, rather than names or addresses, the content of a message is used to route the message. As a consequence, a publisher does not have to know which will be the receivers of its message. Vice versa, an object interested in a certain content can receive the requested information without knowing the publisher. In this way, communication is anonymous and the producers and consumers are completely decoupled.

Clearly, many of the above mentioned goals for a communication and interaction scheme are met by the P/S model. However, the mapping of the content-based approach to a physical network may not be straightforward. As described in [11], depending on the network structure there are different ways to implement the scheme. Roughly, there is a trade-off between efficiency and flexibility. If the properties of the underlying network are completely neglected, the content-based approach requires that every message is examined by every node to decide whether an object on the node has subscribed to the respective content.

Therefore, event channels are introduced in [11] which allow the late binding of message content to addresses. This binding happens at run time just before communicating the first time, i.e. when subscribing to a content class or when publishing the first time. It should be noted that the way this binding is performed is dependent on the underlying network structure. A more detailed review of possibilities can be found in [11].

3 Case study

To highlight the different properties of the interaction schemes in a realistic scenario, the JANUS robotic system has been adopted as a test bench. JANUS is a stationary robotic system consisting of a vision system and two complex arms as manipulators. The vision system is mounted on a neck with two degrees of freedom. To manipulate or grab objects, JANUS is equipped with two arms with eight degrees of freedom each. All joints are monitored by optical encoders which together with the vision system form a complex sensor network. Figure 1 shows the proposed control architecture which rises the interaction scheme used in the case study.

The distributed model of control used in this example is based on a multi-agent approach [5]. In this model, the system is hierarchically partitioned in two levels:

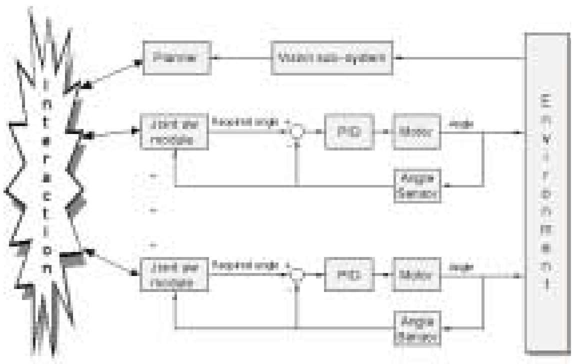

Figure 1. Interaction scheme.

1. a reactive level, where an agent (or object, or component) is associated with each joint, having the sensor and actuator capabilities shown in Figure 1;
2. a planning level, where a planner must communicate with the agents in the reactive level in order to give then directives and goals. For our purposes, a goal is simply a Cartesian position to be reached by the effector, e.g. grabber, mounted at the end of an arm.

Once a plan is ready, it is communicated to all agents compounding a member, as illustrated in figure 1, and the distributed control algorithm is executed. This algorithm will be briefly explained here for the 2 dimensional case (see Figure 2). The cycle starts at the end-effector related to the joints, which minimizes the angle defined by the lines connecting the respective joint and the effector and the line connecting the joint and the Cartesian coordinates of the goal, as in Figure 2 (a). Once this is done, the new position of the joint is communicated to all other joints, and a token must be passed to the neighbor joint. The token circulates through all joints, each one performing the minimization criterion, until the goal is reached. Off course, the algorithm is over simplified here and some additional constraints were taken into account. Focusing on the adopted communication pattern, it should be emphasized that all software modules related to the joints need to exchange information.

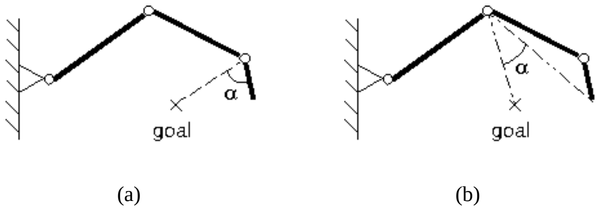

Figure 2. Algorithm description.

In order to highlight the differences between the communication strategies presented in the previous section, the JANUS robot system was implemented using the three alternative and results are presented in the next sub-sections.

3.1 RMI communication model

The general components of JANUS robot system are presented in the class diagram in Figure 3. However, the class diagram does not represent the communication relations that we want to analyze. The communication relations which are necessary during run-time are depicted in the instance diagram of Figure 4. Because the RMI has to establish point-to-point connections between every joint, we obtain a total number of 64 connections for one arm. To improve the figure's readability only connections from the first joint are depicted. The other joints have similar connections.

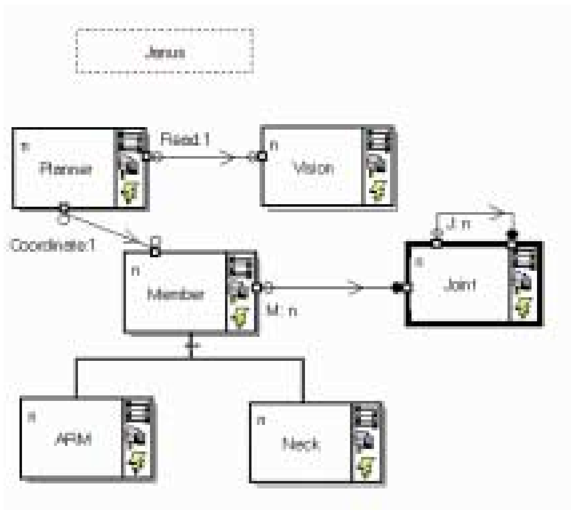

Figure 3. Object-oriented model for the JANUS Robot System.

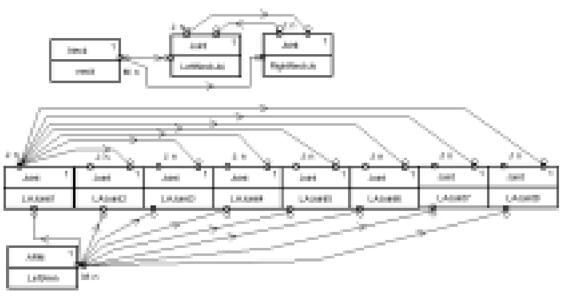

Figure 4. Instances connections in the RMI communication model.

In this communications model each connection represent a sender/receiver pair. If a joint wants to broadcast its new position to the others joints, it explicitly has to send an RMI to each specific joint. In the JANUS approach the arm is moved in steps and each step requires one move for each joint. As a result 56 messages with a similar information and addresses have to be sent. This way, a control plane change will lead to sending hundreds of synchronization messages. Hence, the RMI model of communication is not well suited for this scenario because broadcast, a commonly needed communication pattern in many control applications, has to be explicitly modeled as point-to-point connections by the programmer. Secondly, because references are used to identify the communication target, modules cannot be developed independently. The knowledge of the object (and its interface) to which a communication relation is maintained is required at design time.

3.2 Port-based model

A way to better support modular design and active components from the communication point of view is the introduction of ports. As discussed in subsection 2.2 a port is an abstraction of a communication channel and is part of the object's interface. Rather than defining a communication relation explicitly in terms of object references, a port reference is a declaration of a component in some higher class definition. Thus, in a way it defines what information is communicated over a port rather than which object is the communication target. That means that the binding of the objects involved in a communication relation are deferred from design time to a later stage of system development, e.g. to the time when the system is configured. From a programmer's point of view, a port clearly removes the problem of programming a broadcast as an explicit sequence of RMIs. Additionally, ports support active behavior of an object. Although the class diagram from both models look like the same, the instance diagram (runtime configuration) from the port-based model is characterized by the presence of the virtual port, as depicted in Figure 5. This way, the main difference between them concerns the object implementation. In the RMI, the sender is responsible for providing the messages recipient. In the port-based scheme, the sender does not know the receiver, it just writes to an output port, and all connected input ports will receive this message. Therefore although not explicitly written by the programmer, both models have the same number of send/received messages.

3.3 Publisher/Subscriber model

In the P/S model, the event channel is an explicit system component. Therefore the event channel has to be added

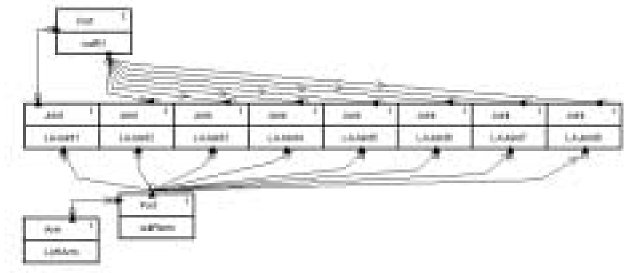
Figure 5. Port-based configuration.

to the class diagram as showed in Figure 6. Also the Figure 7 shows the instance diagram of the JANUS robot using event channels. The advantages for the system designer to the previous modeling techniques become obvious immediately. There is an event channel "LeftArmChannel" which handles all communication in the left arm of the robot (the right arm is modeled correspondingly). All joints publish their positions in this channel. At the same time, all joints have subscribed to messages handled by this channel. The many-to-many communication relation explicit in the RMI model is now collapsed to a single broadcast channel. As a consequence will have just 18 Member to Joints and Joint to Joints connections in the instance diagram. One step movement of the arm represents just 8 messages. As a consequence the number of synchronization messages which need to be modeled for a complete plane move will be one magnitude lower than in the RMI model.

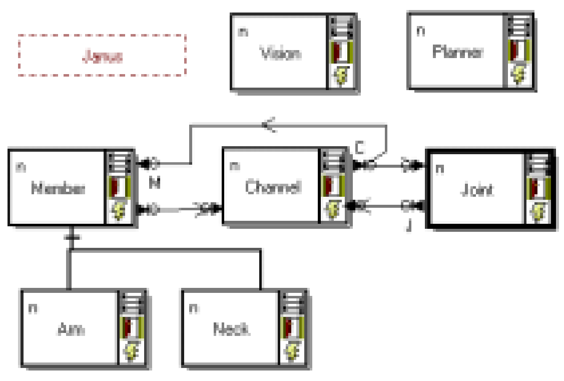

Figure 6. P/S object model.

4 Comparison

The previous section dealt with a comparison from a design level perspective. The main goal was to show that broadcast communication patterns often needed in control systems are not well supported by the RMI model. Let's

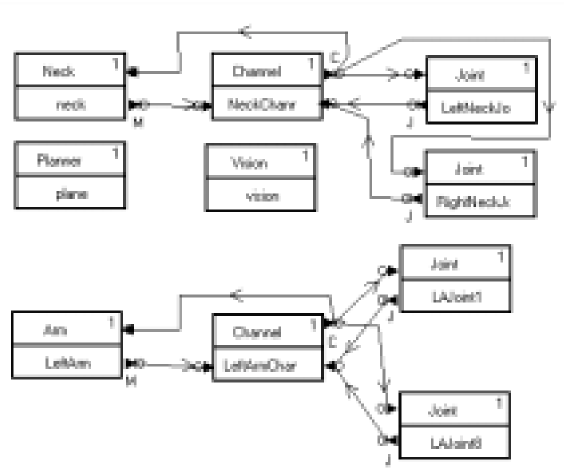

Figure 7. Instances of event channel model.

consider now modularity, extensibility, and autonomy aspects of the communication models.

As described earlier in the paper, RMI usually couples the transfer of data with a transfer of control thus violating the principle of control autonomy which is crucial in a distributed control system. Additionally, when using remote method invocations it is necessary to know which object to address. Thus, the programmer cannot design an object independently from other objects. As a result, any changes in the communication relations will affect a number of objects that may be difficult to trace.

In a port-based communication model, a port just defines outgoing and incoming messages for an object. Thus, the designer of an object does not need to know which other objects are involved in the communication as it is the case when using remote invocation. This supports the important feature of anonymity, i.e. the producer of an information instance does not need to know which are the consumers of this information and vice versa. Anonymity is an additional isolation property that supports modularity and eases modifications. To establish a communication relation between objects, a connection between ports has to be defined in an extra binding step. Usually this binding is performed as part of the configuration when composing the system out of the individual objects. Clearly, this extra step supports modularity, extensibility and modifiability of the system because now the individual objects are not affected or must be modified when changing the communication relations. Concerning autonomy of objects, ports do not force any form of control transfer as it is done in remote invocation.

It should be noted, however, that although ports decouple the design of a component from the definition of its communication relations, connections between ports are statically configured. This means that during run-time communication relations cannot be changed, added or removed dynamically. Sometimes however, it is necessary to provide uninterrupted control, which requires on-line modifications or extensibility of the system [2]. If this is the case, communication relations have to be changed or added during run-time. As described earlier, the publisher/subscriber communication model has been designed to meet this requirement. In its most general form it uses a broadcast to all sites that then have to examine the complete stream of messages to filter those messages which are needed locally. Thus, this mechanism provides a content-based routing scheme. It avoids any binding at the price of a substantial overhead for the filtering function. The introduction of event channels that bind a message content to a network address overcomes this problem. Event channels are in many respects similar to ports. The difference is that the binding is dynamically performed during run-time when an object wants to communicate the first time. Therefore the binding has not to be specified at any instance during the design or configuration process. Clearly, dynamic binding needs some effort during run-time what is to a certain extent dependent on the underlying network structure. In [13] a distributed set of IPC-demons keep track of the binding between channels and network addresses. In [11] a central broker holds the binding tables. Whenever an object communicates via an event channel the first time, this event channel broker is involved in resolving the channel to address binding. It should be noted that the binding has only be performed once and does not necessarily happen in the real-time loop or the "steady state path" [13]. The binding could e.g. be performed in a initialization phase for a new component when it is integrated in the system. The properties discussed along the article are summarized in table 1, which allows a direct comparison from the different communication schemes.

Table 1. Summary of communication characteristics.

	RMI	Port	Content	Channel
model	client-server	producer-consumer	producer-consumer	producer-consumer
routing	obj. name or address	port name or address	message content	channel type
binding	design time	configuration time	no binding	run time
control transfer	yes	no	no	no
topology	point-to-point	point-to-point	broadcast	multicast-broadcast
filtering	sender	sender	receiver	receiver

5 Experimental results

In order to evaluate the temporal characteristics of the communication schemes being compared, some measurements based on three different implementations have been performed. To eliminate the variation that could arise from different basic implementations, the three schemes were implemented from scratch using connectionless communication with datagrams as the underlying communication mechanism. Therefore it can be assured that the communication workload was tantamount to the maximum extent in the three cases. Although the UDP protocol of connectionless datagrams does not support any kind of end-to-end QoS assurance, a good understanding of the empirical communication characteristics can be achivied if access to the communication medium is rigorously controlled and the data produced is properly analyzed with statistical tools.

The testbed was an isolated network consisting of 4 PCs Pentium 133 MHz fully interconnected through an Ethernet medium. Each *joint* instance ran as an active object (i.e., in its own thread of control) and the load was equally distributed among the hosts, i.e., two *joint* instances per host. It should be remarked that as long as the implementation is concerned, there are minor differences in the load between RMI and event channels on the one side and the port-based scheme. RMI and P/S models have an extra object participating in the execution of the algorithm: the *arm* object in the case of RMI and the *event channel* object in the case of P/S. Although the *arm* object is present in the Port model (aggregating the *joints* and sending objectives to them when necessary), it does not participate in the execution of the algorithm as long as in this case the control is transferred directly from one *joint* instance to the next in the chain.

At each round of the applied distributed algorithm, a chain of events is triggered, which propagates from *joint 0* to *joint 7*. The end-to-end requirements related to the occurrence of the trigger event at *joint 0* and the end of the action at *joint 7* related to this has been chosen for empirical comparison of the schemes. The setup for the experiments exist of defining one of the active objects as trigger, which is executed as a periodic process. The other objects just respond in the chained events triggered by the periodic one. For example, in the case of port-based and event channel implementations, *joint 0* was such triggering object, while in the RMI implementation, the *arm* played this role. System calls were included in the code (the load of this system call was determined to be 2 μsec in the worst case) to measure the time elapsed from the beginning of the round to its conclusion on the last *joint* of the chain, taking care to run both objects (the trigger and the completion one) in the same host to avoid the necessity for clock synchronization.

From the experimental data it was possible to determine that the time periods measured followed a normal distribution with the means and the standard deviations depicted in table 2. The sample size for each experiment was 3000 measurements and for a confidence level of 90% this resulted in the confidence interval for the mean estimation presented in the third column of the table.

Table 2. Results from experimentation.

	mean	stand. dev.	conf. interv.(90%)
RMI	535 milisec	45 milisec	$533.6 < \mu < 536.4$
Port	485 milisec	68 milisec	$482.9 < \mu < 487.1$
P/S	536 milisec	75 milisec	$533.7 < \mu < 538.3$

The data is in compliance with the analysis and the considerations about the implementation above. Firstly, it is showed that for distributed applications with intensive broadcast, the three schemes have very similar end-to-end requirements. Secondly, the slightly better results for the port-based scheme are a consequence of the fact that RMI and the event channel model need one additional participating object, as explained above.

6 Conclusions and future work

The paper has compared different strategies for interacting and communicating distributed real-time objects. From the developed case study one can clearly observe that the remote method invocation concept, the most frequently adopted communication model in distributed object-oriented systems has several drawbacks when compared to other approaches. Not only the fact that communication binding must be done explicitly at design time , thus decreasing the re-usability of the same class in different contexts, the adopted point-to-point communication leads to ineffective way of communication in distributed control systems.

Further time measurements of the implemented alternative solutions for the JANUS case study are being performed. They will allow a comparison of the obtained real-time properties, such as execution time, cyclical activation jitter, and specially the differences in the temporal behavior caused by the overhead imposed by the different communication schemes.

References

[1] D. R. A. Carzaniga and A.L.Wolf. Achieving scalability and expressiveness in an internet scale event notification service. In *Nineteenth ACM Symposium on Principles of Distributed Computing (PODC2000),* Portland OR., July 2000.

[2] A. S. B. Oki, M. Pfluegl and D. Skeen. The information bus-an architecture for extensible distributed systems. *14th ACM Symposium on Operating System Principles, Asheville, NC*, pages 58–68, December 1993.

[3] B. G. B. Selic and P. Ward. *Real Time Object Oriented Modelling*. John Wiley and Sons, Inc., 1994.

[4] A. Birrell. Implementing remote procedure calls. *ACM Transactions on Computers Systems*.

[5] P. Bohner and R. Lppen. Redundant manipulator control based on multi-agents. In *3rd IFAC Symp. on Intelligent Components and Instruments for Control Applications*, pages 357–362, June 1997.

[6] N. Carriero and D. Gelernter. Linda in context. *Commun. of the ACM*, 32(4):444–458, April 1989.

[7] R. G. D. Estrin and J. Heidemann. Scalable coordination in sensor network. In *Proc. ACM/IEEE MobiCom*, 1999.

[8] R. V. D.B. Stewart and P. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *IEEE TC on Software Engineering*, 23(12), December 1997.

[9] W. R. J. Kulik and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. of the ACM/IEEE MobiCom*, 1999.

[10] Javasoft. Java rmi enhancements since jdk 1.2. Technical report, Javasoft, www.javasoft.com/products/jdk/1.2/docs/guide/rmi/index.html, 1999.

[11] J. Kaiser and M. Mock. Implementing the real-time publisher/subscriber model on the controller area network (can). In *Proceedings of the 2nd Int. Symp. on Object-oriented Real-time distributed Computing (ISORC99), Saint-Malo, France*, May 1999.

[12] K. Mori. Autonomous decentralized systems: Concepts, data field architectures, and future trends. *Int. Conference on Autonomous Decentralized Systems (ISADS93)*, 1993.

[13] M. G. R. Rajkumar and L. Sha. The real-time publisher/subscribe inter-process communication model for distributed real-time systems: Design and implementation. *IEEE Real-time Technology and Applications Symposium*, June 1995.

[14] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, London, 1999.

[15] S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Massachusetts, 1999.