Proceedings of the 2002 IEEE/RSJ
Intl. Conference on Intelligent Robots and Systems
EPFL, Lausanne, Switzerland • October 2002

A Software Framework based on Real-Time CORBA for Telerobotic Systems

S. Bottazzi, S. Caselli, M. Reggiani, M. Amoretti

RIMLab – Robotics and Intelligent Machines Laboratory
Dipartimento di Ingegneria dell'Informazione, University of Parma, Italy
{bottazzi, caselli, reggiani, amoretti}@ce.unipr.it

Abstract

The technological developments in distributed systems have led to new telerobotic applications, such as virtual laboratories and remote maintenance of complex equipment. These applications must satisfy both the general requirements of distributed computing, e.g. location transparency and interoperability, and the domain-specific requirements of reconfigurability, guaranteed performance, real-time operation, and cooperation among robots and sensory systems. In this paper, we describe a software framework for distributed telerobotic systems exploiting advanced CORBA features, including Asynchronous Method Invocation and real-time priorities. The framework allows development of portable multithreaded client-server applications supporting concurrent and preemptable actions in the target robot system, and has been evaluated in a laboratory setup including a robot manipulator and two cameras accessible by multiple clients.

1 Introduction

The traditional approach to robot teleoperation [17] is based on a dedicated system architecture ensuring tight coupling of a master device and a slave robot by means of a dedicated connection. This approach, though, is unable to cope with novel remote operation paradigms such as tele-teaching/tele-learning, virtual laboratories, on-line robots, and projects requiring collaboration among geographically distributed users [1, 2, 9].

On top of a seemingly conventional telerobotic system architecture (Figure 1), these new applications require support for distributed collaboration among multiple sites, dynamic reconfiguration of physical robotic resources and user connections, up to public outreach in media-covered experiments [2]. In order to make them available to multiple users, physical resources, such as robots or sensors, must be managed by software applications (termed *servers*) accepting incoming requests and providing control and arbitration over resource allocation. Users interact with *client* applications for task programming and monitoring. A key feature of the system architecture in Figure 1 is the interconnection channel, which now must be shared among multiple clients and servers to enable distributed collaboration. When users have arbitrary locations, or when distance and cost factors dictate it, the interconnection channel is implemented through the Internet.

The use of standard, general-purpose interconnection technologies, while providing essential functional advantages, exacerbates the latency and bandwidth problems that must be faced in all telerobotic systems. These problems compound with certain properties typical of new applications. In distributed telerobotic applications, systems are extremely *heterogeneous*: they are often implemented using previously available devices, based on hardware acquired from different vendors, running different operating systems and programmed in a variety of languages. Moreover, these systems are also *dynamic*: in a general scenario, sensor and robot controllers can dynamically connect to the network resulting in changes in the number and location of peers, in service roles that can change at run-time, and in the need of load reconfiguration to improve distributed system performance. Clients can also register or disconnect at run-time and bid for commanding or supervisory roles, leading to a variety of dynamic interaction patterns.

Building a teleoperation architecture from scratch for these heterogeneous and dynamic systems is not always feasible, due to economic and time constraints. Following a trend in modern distributed systems design, open, reconfigurable, and scalable architectures can be built using commercial off-the-shelf (COTS) components. The object-oriented (OO) design methodology provides fundamental concepts such as inheritance, polymorphism, and hiding, useful in the development of complex distributed COTS-

based applications [6]. In this area, exploitation of middleware software between low-level APIs and client/server applications is a major approach pursued to cope with dynamic and flexible applications. Several established middleware implementations are based on the Common Object Request Broker Architecture (CORBA) (`http://www.corba.org`), a vendor-independent specification promoted by the Object Management Group (OMG) (`http://www.omg.org`). CORBA overcomes the interoperability (alias heterogeneity) problem mentioned above, since it allows integration of systems built using different software technologies, programming languages, operating systems, and hardware.

A middleware such as CORBA is simply a tool for connecting objects across heterogeneous processing nodes, but does not implement an application by itself. Indeed, additional work is required to develop a telerobotic application, which is not a standard client/server application. Implementing servers is well-known to be a complex and time-consuming task, as a number of specific constraints must be met. In a telerobotic system, a server should satisfy the following requirements:

- have a multithreaded structure with concurrency mechanisms,
- allowing an improvement in perceived response time, and
- simplifying sharing of CPU among computational and communication services.

It should be able to deal with client requests preserving their ordering, and exhibit different kinds of reaction depending on their urgency. It should operate in real-time to allow implementation of the appropriate control laws with guaranteed operation. Finally, a server should provide synchronization mechanisms for exclusive allocation of non-sharable resources. Indeed, until the recent extension of CORBA [13], no standard API for multithreading, synchronization, or asynchronous calls was available, forcing developers to build dedicated solutions or to rely on proprietary features.

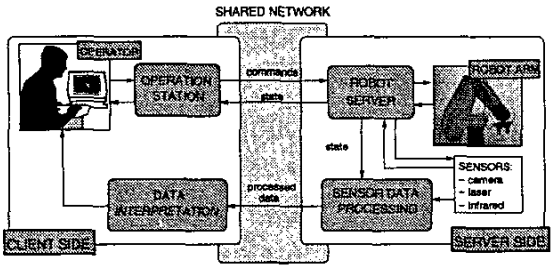

Figure 1: Components of a telerobotic system.

In this paper we describe a framework exploiting the latest CORBA specifications to meet the requirements of modern telerobotic systems. Section 2 provides a background on advanced CORBA features, and more specifically on the features exploited in the framework proposed in this paper. Section 3 describes the proposed general framework for distributed telerobotic systems. The experience collected so far in implementing the framework in a laboratory setup is reported in Section 4. Section 5 discusses related work and Section 6 summarizes the paper.

2 Advanced CORBA features

Since its inception as a standard, CORBA has supported interoperability among heterogeneous systems, providing a flexible communication and activation substrate for distributed computing environments. A detailed description of CORBA basic components can be found in [10]. *Internet Inter-ORB Protocol* (IIOP), *Interoperable Object Reference* (IOR) *Interface Definition Language* (IDL), *CORBA Services* [13] are some of these components that allow CORBA to be successfully exploited in many "standard" client/server applications. This section summarizes new features of the CORBA Standard 2.4 [13] and their impact on telerobotic applications. With these extensions, COTS middleware can be used even in critical applications, requiring strict control over allocation of CPU and memory resources.

A telerobotic system should provide the operator with the guarantee of correct execution priorities of application tasks at the server. The peer heterogeneity of many applications precludes the use of a common priority scale, forcing users of earlier CORBA versions to concern about low-level details of threads on different OSes. With the new RT CORBA specification [7], developers can rely on portability of end-system priority. A new feature, **priority mapping**, converts CORBA priority levels assigned to CORBA operations to OS native priority levels or vice versa.

Teleoperation applications also require that the task be executed at the right priority on the server side. RT CORBA allows one of two models to be adopted on a per-method invocation basis: `SERVER_DECLARED` and `CLIENT_PROPAGATED`. In the `SERVER_DECLARED` model the server defines the execution priority when object methods are invocated, whereas in the `CLIENT_PROPAGATED`, exploited in the proposed framework, the client establishes the prior-

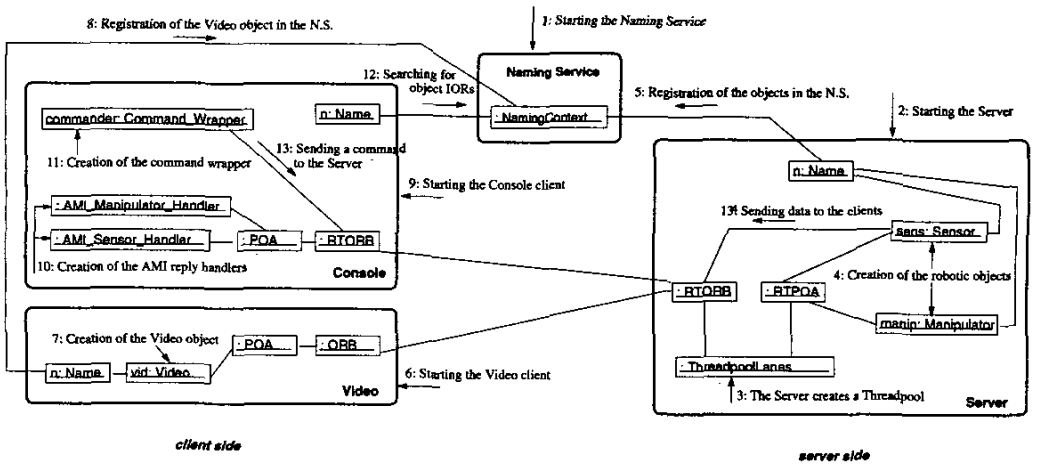

Figure 2: *Collaboration diagram of the teleoperation system based on CORBA. The first service available is the Naming Service (1), providing location transparency. Next, servers (2-5) and clients (6-12) are created and communication among peers can begin (13-14).*

ity of the operation, and this priority must be honored by the server.

Control of several robots and sensors teleoperated from multiple remote clients requires a multithreaded server allowing concurrency among actions. Moreover, the server should be able to discriminate among services, granting privilege to critical tasks (emergency stop, reading of proximity sensors), and should avoid priority inversion, with low-priority tasks blocking high-priority ones.

To support programming of multithreaded servers, RT CORBA provides **Thread Pool**, a mechanism enabling preallocation of server resources. With the Thread Pool mechanism, a group of threads is statically created by CORBA at the server at start-up time. These threads are always ready to be bound to requested methods, while a fixed cap is set for dynamic threads, which are created only once static threads are exhausted. Thread Pools avoid the overhead of thread creation/destruction at run-time and help in guaranteeing performance by constraining the maximum number of threads on each host.

Under the (extreme) condition where the whole set of threads has been bound to low-level requests, the server could miss the deadlines of high-priority actions, a situation clearly unacceptable in a robot teleoperation system. To avoid depletion of threads by low-priority requests, a Thread Pool can be further partitioned in **Lanes** of different priority. This partitioning sets the maximum concurrency degree of the server and the amount of work that can be done at a certain priority. Partitioning in Lanes and related parameters cannot be modified at run-time; the only freedom is reserved to higher priority methods which can "borrow" threads from lower level Lanes once their Lane is exhausted.

RT CORBA allows consistent synchronization methods semantics between CORBA applications and middleware by means of the **Mutex** interface, implementing the mutual exclusive lock. The Mutex interface is the basic synchronization mechanism and improves portability and correctness of application code, however RT CORBA still lacks higher level synchronization mechanisms such as condition variables, semaphores, and barriers. Moreover, as no standard API for the priority inversion protocol is prescribed by RT CORBA, the user must rely on ORB-specific APIs to fully prevent priority inversion [15].

An additional advanced feature exploited in the proposed framework is drawn from the CORBA 2.4 Messaging specification [13]. Standard service requests in CORBA systems rely on the Synchronous Method Invocation (SMI) model, that blocks the client until the server notifies the end of the requested activity. This approach is acceptable for simple teleoperation applications consisting in the stepping of one action at a time, possibly in stop-and-go mode, whereas SMI is clearly unsuited for more advanced telerobotic scenarios where the user can invoke execution of multiple concurrent actions at the server. Examples of such tasks are coordinated operation of multiple arms or concurrent sensing and manipulation. Non-blocking invocations with earlier CORBA versions either relied on methods not guaranteeing the delivery of the request or on techniques requir-

ing significant programming efforts and known to be error prone.

A more efficient way to perform non-blocking invocations has been made available in the recent extension of the CORBA Messaging through the **Asynchronous Method Invocation (AMI)** model, with either a polling or a callback approach. Moreover, as AMI and SMI share the same object interface, clients can choose between synchronous or asynchronous calls while server implementation is not affected. AMI allows a CORBA-based system to efficiently activate multiple concurrent actions at a remote teleoperated side, as shown in section 3.4.

3 Framework

This section describes the proposed general framework for distributed telerobotic systems. The collaboration diagram in Figure 2 provides a snapshot of the main components and their interaction.

3.1 Robot server

The implemented robot server allows concurrent control of robots and sensors. It receives commands, executes them, and sends results or sensor data back to clients. The server application is multithreaded and sensitive to priority of requests. At start-up time, the server creates a Thread Pool with Lanes. Threads in every Lane take care of requests in a range of priorities using the Leader/Followers pattern [16]. Then, the server associates the Thread Pool to the *Real Time Portable Object Adapter* (RTPOA) [13]. The server process uses the RTPOA to create/delete CORBA objects for every controlled robot or sensor, and to dispatch the incoming requests to the *CORBA Servants*, the object concrete implementations. Finally, registration of created robot/sensor objects in the Naming Service provides location transparency. Adding support for a new device (either a robot or a sensor) in the server simply requires to write the interface of the methods that will be available to clients and their implementation (Figure 3). Location transparency, multithreaded control, command concurrency, and priority mapping are already provided by the server implementation. Moreover, moving the device to another robot server does not need any change in the code. The robot server code is fully portable to a number of operating systems that support multithreading and real-time, including QNX, VxWorks, and several general purpose OSes.

3.2 Distributing sensor data

Sensor data are distributed to clients using the Observer pattern [8]. This pattern requires definition of two CORBA classes for each available sensor: the *subject* at the server side and the *observer* at the client side. An example of the implemented solution for a video CCD camera follows.

To receive video data from a camera, the client calls the method `attach(Video x)` on the Camera object (the *subject*), passing a reference to a Video object (the *observer*) (Figures 2 and 3). Each Camera object in the server holds a list of all Video objects that have been attached. When video data are ready, the server can invoke the `video_obj[i].draw(image_data)` method of all "attached" video objects. The method `detach()` allows the client to stop the video stream, removing its reference from the list. Data acquisition and distribution are synchronized using blocking CORBA calls; as no polling operation is used, this technique is very efficient.

New sensors can reuse the same code; therefore their implementation only requires the definition of the data type.

3.3 Robot client

Users can define application tasks either submitting a sequence of single commands (i.e., move the arm to position P_i, get robot position, start grabbing a video stream, ...) or by means of a program that will then be interpreted. When the control flow is a sequence of single actions, the client can use Synchronous Method Invocations to a remote object method. When the flow involves a set of actions to be run in parallel, the robot client uses the Asynchronous Method Invocation model. In order to use AMI on a remote object, a client must define an AMI handler for it (item 10 in Figure 2) so that the ORB can manage server replies related to previous AMI calls.

Additional features of the current robot client implementation are location transparency of controlled devices, whose reference is available using the Naming Service, transparency to servers OSes, thanks to RT CORBA services such as priority mapping, and possibility to establish priority of tasks, using the `CLIENT_PROPAGATED` model.

3.4 Concurrency in action execution

Asynchronous Method Invocation (AMI) offers a powerful tool to support parallel execution of actions at the server side. When the server receives non blocking requests from the client, it dispatches them to the Thread Pool according to their priorities.

A common approach to the execution of parallel actions is to start them as soon as possible, i.e. when their requests reach the server. However, in robotics applications a set of parallel actions must often begin at the "same" time, as the coordination of their

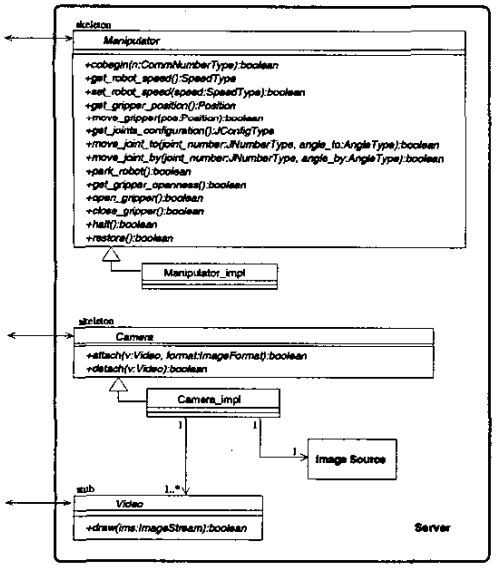

Figure 3: *The modularity of the framework eases integration of new objects (robots, sensors) in a server. It is only required to define the IDL interfaces and the implementation of exposed methods (actions available to clients). The class diagram shows the IDL interface for a manipulator and a camera.*

execution is required to ensure logical correctness or safety. This is the rationale for the introduction of *waiting rendezvous* strategy [6] in the framework. An instruction termed `cobegin(n)` prefixes the invocation of parallel actions on a server. The `cobegin(n)` acts as a barrier for the next n method invocations, whose execution, therefore, does not start until all calls have reached the server.

To implement `cobegin(n)`, condition variables should be used, a mechanism not yet provided in RT CORBA. For code portability we avoided the use of OS native system calls for multithreading, and implemented the barrier with the simple lock()/unlock() functions available in RT CORBA. This solution is not fully efficient because the waiting threads must wake up periodically to check the barrier state. We remark that the `cobegin(n)` method is not mandatory for parallel execution of actions. Without `cobegin(n)` the server schedules AMI requests as soon as they arrive.

4 Experimental assessment

The implementation of the framework is written in C++ and based on The ACE ORB (TAO) [5], a freely available, open-source, and standard-compliant real-time implementation of CORBA. It is on the edge in supporting RT CORBA, and its

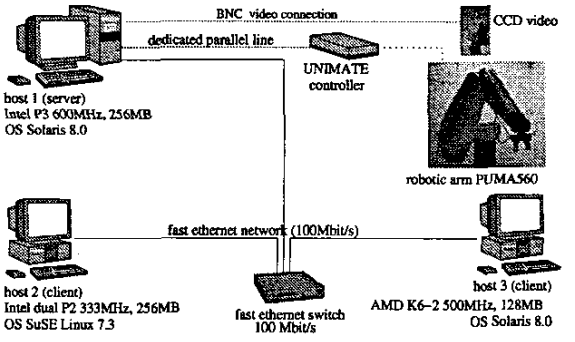

Figure 4: *Experimentation setup.*

proprietary features are often considered for inclusion in the next version of the standard. TAO supports many operating systems, including Solaris, RT-Linux, VxWorks, QNX, and Lynx. The framework described in section 3 has been used to build a telerobotic system and some applications with simulated workload.

4.1 Telerobotic testbed

The teleoperation testbed (Figure 4) is composed of clients and a server running on three machines connected using a Fast Ethernet switch. The server controls three physical devices: a Puma 560 robot arm and two CCD cameras (one wrist-mounted and the other on the ceiling shooting the whole testbed area). The robot server runs on Solaris 8, a general purpose operating system that supports real-time multithreading. Robot clients can run on both Solaris and Linux machines. The controlling client should be located on a Solaris machine in order to set real-time features at the server; monitoring clients can run also on Linux, even with incomplete real-time support.

Several applications have been implemented to test the correctness of the framework and identify the relevant parameters for the robot server. Three Lanes (low, medium, and high priority) have been defined for the Thread Pool. Low and medium priority Lanes supply threads for the execution of actions composing the goal task. The high-priority Lane supplies threads for emergency actions, so as to guarantee their immediate dispatch.The scheduling algorithm at the server is a Priority Level Round Robin (`SCHED_RR`), which is available in any POSIX-compliant operating system. Future versions of RT CORBA standard will likely include dynamic scheduling algorithms, such as Earliest Deadline First, Maximum Urgency First, Minimum Latency First, already available in TAO proprietary exten-

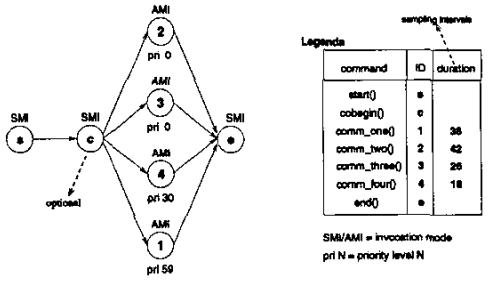

Figure 5: *Precedence graph of a concurrent task, composed of an initial action* `start()` *followed by four concurrent actions with different priority.*

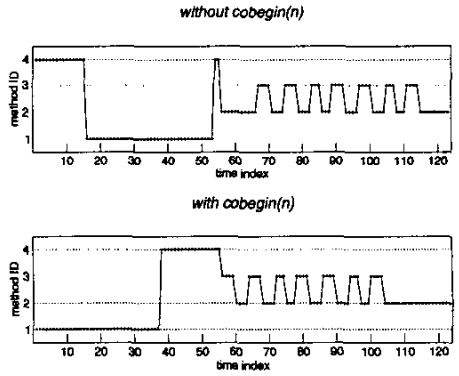

Figure 6: *Experimental results of concurrent service without (top) and with (bottom)* `cobegin(n)`.

sions.

Description of the implemented applications, together with short videos cues, is available at the project page (`http://rimlab.ce.unipr.it/Projects/Teleoperation`).

4.2 Concurrent execution

Several experiments involving simulated workload have been carried out to evaluate the correctness and robustness of the robot server. The robot server has been tested with a variety of sets of concurrent actions, with different priority levels and synchronization. A goal of these experiments is to verify the effectiveness of `cobegin(n)` in avoiding priority inversion in the execution of parallel actions. One of the experiments is described in Figure 5, showing the precedence relations, duration and priority of each method call.

The correct outcome of this experiment requires that the four concurrent methods be executed according to their priority. Figure 6 compares two experimental executions of the task. Without `cobegin(n)` (top diagram), the medium priority action (ID 4), whose request is the first reaching the server, is executed before the high priority action (ID 1). With `cobegin(n)` (bottom diagram), the priority of threads is always guaranteed and no priority inversion occurs.

5 Related work

Our work relates to the area of internet-based telerobotics, whose aim is to build flexible, cheap, dynamic, heterogeneous distributed telerobotic systems and applications. A broad perspective on these applications is given in the collection [9]. The main issue in many of these projects is the interaction with web users who, lacking technical skills, require easy-to-use command interfaces.

Other research views internet-based telerobotics as *distributed robotic systems (DRS)* [4], addressing the issues arising in the implementation of client/server systems. A few papers exploit the interoperability and location transparency provided by CORBA to ease system implementation in applications such as a distributed laboratory [14], a supervisory control scheme [3], or an internet telerobotic system conceived to provide assistance to aged and disabled people [12].

Two recent papers are more directly concerned with the implementation of systems supporting distributed telerobotic applications. Hirukawa and Hara [11] propose a framework based on OO programming for robot control, whereas Dalton and Taylor [4] advocate nonblocking asynchronous communications, viewed as essential to build a distributed robotic systems. Since this feature was not available in the CORBA implementation they relied upon, the architectural framework in [4] exploits non-standard middleware. We believe that with the current CORBA and RT CORBA specifications, including the AMI invocation model and other advanced features described in section 2, this choice is no longer justified.

Our research departs from this prior art in several respects. To our knowledge, we have developed the first telerobotic application based on COTS middleware not merely for interoperability or location transparency, but taking full advantage of its multithreading and real-time features. No previous work in the area has used the Asynchronous Method Invocation model, even though an asynchronous interface is deemed an essential feature [4]. Now that RT CORBA technology has matured, it can be leveraged upon to develop reliable COTS-based telerobotic systems with strict control over scheduling and execution of CPU and memory resources.

6 Conclusions and future work

The technological developments in distributed systems have led to new telerobotic applications, such as virtual laboratories and remote maintenance of complex equipment. The viability and cost effectiveness of these applications rely on features of COTS-based systems such as location transparency and interoperability. Distributed telerobotic applications, however, pose additional challenges such as reconfigurability, guaranteed performance, real-time operation, and cooperation among physical robots and sensory systems.

The recent extensions of the CORBA specification have made it better suited for the needs of distributed telerobotic systems. Our investigation has been an initial, largely positive assessment, even though additional experimentation is needed to generate feedback to middleware designers. For example, CORBA synchronization mechanisms deserve further improvement. Moreover, a common concern about CORBA-based systems is that the end system can be slower, bigger and more difficult to understand. In our experience, this remark is at least partially true; nonetheless, for long-term projects with evolving goals and complex applications, CORBA as a middleware provides far more advantages than inconveniences.

In this paper, we have described a software framework for distributed telerobotic systems exploiting advanced RT CORBA features to achieve flexibility, portability, openness, extensibility, and reusability of the application. The AMI interface and real-time features allow development of a portable multi-threaded server supporting concurrent and preemptable actions. We are currently developing (in Java) an advanced interface enabling skilled users to graphically design tasks including concurrent and conditional actions.

An open question in distributed telerobotic applications is the partitioning of control over task execution between clients and servers, and the degree of autonomy that should be left to servers to overcome latency and bandwidth limitations.

Acknowledgments

Work on this paper has been supported in part by MURST, Italy (Project ISIDE, *Dependable Reactive Computing Systems for Industrial Applications*) and by CNR, Italy (Project No. CNRC00FE3_001 *Robot system architectures for distributed virtual laboratories*).

References

[1] *WS2001: Int. Workshop on Tele-Education in Mechatronics Based on Virtual Laboratories*, Weingarten, Germany, July 2001.

[2] P. Backes, K. Tso, and G. Tharp. Mars Pathfinder Mission Internet-Based Operations using WITS. In *IEEE Int. Conf. on Robotics and Automation*, Leuven, Belgium, May 1998.

[3] R. L. Burchard and J. T. Feddema. Generic Robotic and Motion Control API Based on GISC-Kit Technology and CORBA Communications. In *IEEE Int. Conf. on Robotics and Automation*, Albuquerque, NM, April 1997.

[4] B. Dalton and K. Taylor. Distributed Robotics over the Internet. *IEEE Robotics & Automation Magazine*, 7(2):22–27, June 2000.

[5] Distributed Object Computing (DOC) Group. Real-time CORBA with TAO (The ACE ORB). http://www.ece.uci.edu/~schmidt/TAO.html.

[6] B.P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.

[7] V. Fay-Wolfe, L. C. DiPippo, G. Cooper, R. Johnston, P. Kortman, and B. Thuraisingham. Real-time CORBA. *IEEE Trans. on Parallel and Distributed Systems*, 11(10):1073–1089, October 2000.

[8] E. Gamma, R. Helm, R.E. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] K. Goldberg and R. Siegwart, editors. *Beyond Webcams: an Introduction to Online Robots*. The MIT Press, 2001.

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

[11] H. Hirukawa and I. Hara. Web-Top Robotics. *IEEE Robotics & Automation Magazine*, 7(2):40–45, June 2000.

[12] S. Jia and K. Takase. An Internet Robotic System based Common Object Request Broker Architecture. In *IEEE Int. Conf. on Robotics and Automation*, Seoul, Korea, May 2001.

[13] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.4*, October 2000.

[14] C. Paolini and M. Vuskovic. Integration of a Robotics Laboratory using CORBA. In *IEEE Int. Conf. on Systems, Man, and Cybernetics*, Orlando, FL, October 1997.

[15] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time Synchronization Protocols for Multiprocessors. In *IEEE Real-Time Systems Symposium*, Huntsville, AL, December 1988.

[16] D.C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley and Sons, 2000.

[17] T. B. Sheridan. *Telerobotics, Automation, and Human Supervisory control*. MIT Press, Cambridge, MA, 1992.