Appears in: *Proc. of the 2003 Intl. Symposium on Distributed Objects and Applications, November 3-7, 2003 - Catania, Italy*

Designing telerobotic systems as distributed CORBA-based applications

Michele Amoretti, Stefano Bottazzi, Monica Reggiani, and Stefano Caselli

RIMLab - Robotics and Intelligent Machines Laboratory
Dipartimento di Ingegneria dell'Informazione, University of Parma,
Parco Area delle Scienze, 181A – 43100 Parma, Italy
{amoretti, bottazzi, reggiani, caselli}@ce.unipr.it
http://rimlab.ce.unipr.it

Abstract. Virtual laboratories and on-line robots are examples of distributed telerobotic systems based on emerging Internet technologies. Building these applications from scratch is a very demanding effort because they must satisfy a wide set of requirements, arising from both the distributed systems domain, e.g. location transparency and multiplatform interoperability, and the telerobotic domain, e.g. guaranteed quality of service, real-time operation, dynamic reconfigurability, concurrent or collaborative interaction among distributed sites. For these systems exploitation of an Object Oriented standard middleware like CORBA should be very effective, thanks to its well known features and services and its recent enhancements (Real-Time CORBA, AMI).
In this paper we summarize our experience in the development of a software framework for telerobotics based on Real-Time CORBA. The framework takes advantage from CORBA services to allow implementation of advanced teleoperation systems, thereby avoiding proprietary or ad-hoc solutions for communication and priority management. In order to enable distributed collaboration and virtual laboratories, it also supports concurrent control and data distribution with multiple Clients. The framework has been evaluated in a real scenario, building a distributed telerobotic application which allows control of a robot arm and several sensors by multiple Clients.

1 Introduction

Internet technology is rapidly evolving, providing access to timely data from news, sports, and financial sources, live video for teleconferencing, and so on. Once we are comfortable with live viewing, the next step is to reach out and touch, by controlling a physical device.

For many years, the pleasure of operating a robot has been limited to trained specialists. Ideally, the Internet opens the door to a much wider audience, but even if many networked robot systems have been developed, these are usually uncommunicative between each other. The main cause is that each system, in

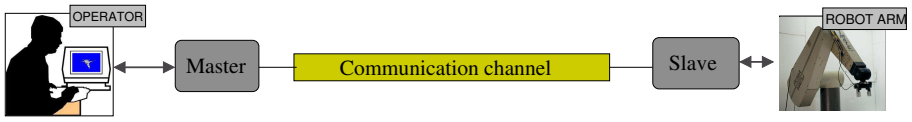

Fig. 1. Traditional model of a teleoperated system.

terms of software architecture, is implemented for itself and from scratch, without using a standard base.

Considering robots (but also sensors, controllers) as objects [1], and networked robots as distributed objects [2] has been the first step towards the idea of open, reusable and scalable software architectures for teleoperation. Distributed object computing is one of the latest research and development topics in computer science and software industry. There are few de facto standards for inter-object communication, e.g. DCOM (Distributed Component Object Model), Java RMI and CORBA (Common Object Request Broker Architecture). By adhering to these standards, development time can be shortened and newly developed components can easily be connected with existing resources.

The vehicle for our research in "distributed robotics" is a CORBA-based software framework we have developed to simplify the implementation of flexible, portable, extensible and reusable telerobotic applications, and whose initial features have been described in [3]. Departing from other works [2, 4–6], we exploited advanced CORBA features to address real-time requirements, concurrency in resource access and in task execution, and data distribution.

In this paper, we discuss the requirements of modern telerobotic systems (Section 2) and the available alternatives in terms of middleware software (Section 3). We next illustrate our enhanced CORBA-based framework (Section 4), describe a distributed application based on the framework (Section 5), and report the experimental results obtained by testing different communication models (Section 6). Finally, we describe some prior work in the field (Section 7), and draw some conclusions (Section 8).

2 Requirements for robotic teleoperation systems

With a brief and intuitive definition, robotic teleoperation allows an operator at one location to perform a physical task at some other location by means of a robotic device [7]. Regardless of the robot autonomy degree and of the distance between sites, a teleoperation system always has three specific components: a command and sensorial interface at the "master" station, used by the human operator to control the task, a "slave" robotic device, which performs actions at the remote site following operator's instructions, and a communication channel between sites. This is the traditional approach to robot teleoperation [7, 8], graphically described in Figure 1.

Telerobotic systems have been typically designed for critical scenarios that are too dangerous, uncomfortable, or expensive for humans to perform. Some ap-

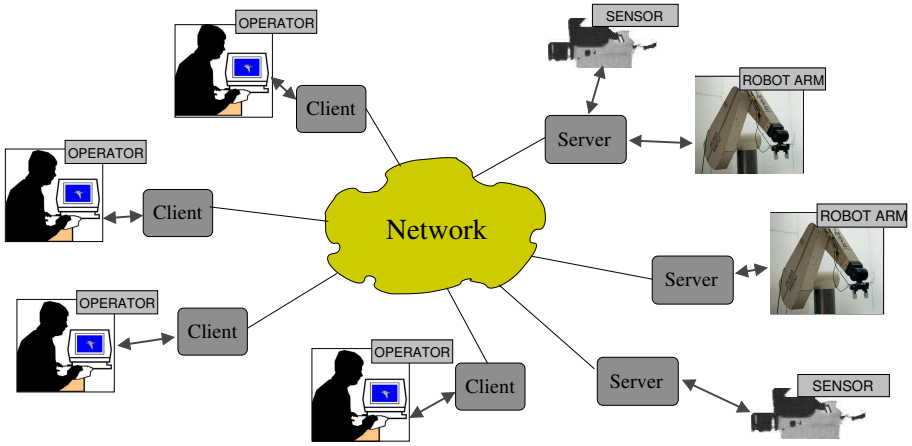

Fig. 2. An advanced teleoperation scenario.

plications are assembly, maintenance or exploration in hazardous environments like underwater, space, nuclear or chemical plant, remote presence, power line maintenance.

In recent years distributed computing systems and the Internet have opened new, broad application perspectives to robot teleoperation systems. Examples of novel applications are tele-teaching/tele-learning, virtual laboratories, remote and on-line equipment maintenance, and projects requiring collaboration among remote users, experts, and devices [2, 9–15].

Although the conventional architecture of Figure 1 still has a fundamental role in a high level description of a telerobotic system, it seems unsuitable for coping with these novel paradigms, which provide new interesting opportunities but also propose alternative challenges.

Modern telerobotic systems (Figure 2) tend to be very dynamic, with users and physical resources, like sensor and robot controllers, which need to be connected/disconnected at run-time to the system (e.g. students or researchers from geographically distributed sites that want to access equipments which are only part-time available, in a virtual laboratory), resulting in changes in the number and location of peers (applications which can act as a Client, as a Server, or both) and in the need of preserving performance of services also with dynamically variable system load.

Operators may have different access levels (from simple sensor monitoring to system supervision), peers must be light-weight and runnable on several platforms (from desktop workstation to embedded system), without increasing applications complexity. Support for different robot programming methods (on-line, off-line, hybrid) should be provided, along with transparency (i.e., all accessible applications must appear as if they were local). Peers directly interfaced with sensors and devices prevalently act as Servers; to execute multiple commands at the same time (from one or several Clients) they should offer a high degree

of internal concurrency. Increasing the number of Clients should not affect system performance (i.e. scalability). Real-time features are needed to preserve QoS and reliability. There is also the need to distribute increasing quantities of sensory data to a potentially large number of Clients during system operation. In a telerobotic task, the timely availability of adequate sensory data to emulate the operator's physical presence at the remote site is particularly crucial [7, 16].

Furthermore, new telerobotic applications often become viable in the context of an existing infrastructure or a constrained budget, which leads to the development of heterogeneous systems built by integrating a mix of new and legacy equipment, based on hardware acquired from multiple vendors, running different operating systems and programmed in a variety of languages. The expensive development of a new application can thus be avoided relying on previous experience or, even better, on a common framework.

3 Middleware software

In the previous section, a list of features and requirements for advanced telerobotic systems has been reported. Building such type of teleoperation architecture from scratch for heterogeneous and dynamic systems is often too demanding, due to economic and time constraints. Following a trend in modern distributed systems design, open, reconfigurable, and scalable architectures can be built using standard middleware software for distributed object computing. Available solutions include JavaSoft's Java Remote Method Invocation (Java RMI), Microsoft's Distributed Component Object Model (DCOM) and OMG's Common Object Request Broker Architecture (CORBA).

Sun's Java RMI (`http://java.sun.com/products/jdk/rmi`) provides a simple and fast model for distributed object architecture. It extends the well-known remote invocation model to allow the shipment of objects: data and methods are packaged and shipped across the network to a recipient that must be able to unpackage and interpret the message. The main drawback of the RMI approach is that the whole application must be written in Java. This constraint is troublesome in common heterogeneous environments of robotic applications, often incorporating legacy and specialized hardware and software components.

Microsoft's DCOM (`http://www.microsoft.com/com/tech/DCOM.asp`) supports distributed object computing allowing transparent access to remote objects. While DCOM overcomes RMI reliance on Java using an Object Description Language to achieve language-independence, it still has limitations concerning legacy code and scalability of applications. Developer's options are indeed restricted because DCOM is a proprietary solution mainly working on Microsoft operating systems.

When language, vendor, and operating system independence is a goal, CORBA (`http://www.corba.org`) is a mature solution that provides similar mechanisms for transparently accessing remote distributed objects while overcoming the interoperability problems of Java RMI and DCOM. Moreover, its advanced and recent features (Real-Time CORBA, AMI) provide functionalities almost essen-

tial in telerobotic applications. At the moment, CORBA seems the logical choice for building complex distributed telerobotic applications.

4 A framework for teleoperation based on CORBA

A middleware such as CORBA offers a set of facilities and tools for connecting objects across heterogeneous computational nodes, thereby simplifying the development of distributed applications. In spite of this fact, development of a telerobotic application remains a fairly demanding task.

We have developed a framework which aims to provide a flexible and effective infrastructure for advanced telerobotic systems. The framework offers handles for implementation of multithreaded Servers, with concurrency mechanisms to simplify sharing of CPU among computation and communication activities, for management of client requests with order preservation and for adaptation of reactions to different requests depending on their urgency. Servers operate in real-time to allow implementation of the appropriate control laws with guaranteed operation. Furthermore, the framework provides synchronization mechanisms for exclusive allocation of non-sharable resources, and three methods of data distribution among peers.

A telerobotic system built with the framework exhibits location transparency, easily achieved through the CORBA middleware. Furthermore, the portability of the framework to various operating systems allows reusability of the same code in several Client stations and reallocation of sensoriality among heterogeneous Server stations. The class diagram in Figure 3 shows the structure of the framework, which is designed to cope with a wide range of situations, providing abstract classes for robots (mobile or not) and sensors. A concrete class implementation requires only adding code for the specific device or controller, whereas the functionalities for communications, concurrency, real-time are shared among all classes and based on CORBA.

In the following subsections we discuss the CORBA features exploited and their usefulness in the telerobotic application domain.

4.1 Real-Time features

A telerobotic system should provide the operator with the guarantee of correct execution priorities of application tasks at the Server. The **Real-Time CORBA** specification [17] provides the developer with handles for resource management and predictability.

The heterogeneity of nodes, in distributed 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. The Real-Time CORBA **priority mapping** mechanism converts CORBA priority levels, assigned to CORBA operations, to OS native priority levels (and vice versa).

Teleoperation applications also require task execution at the right priority on the Server side. RT CORBA defines two invocation models: `SERVER_DECLARED`, in

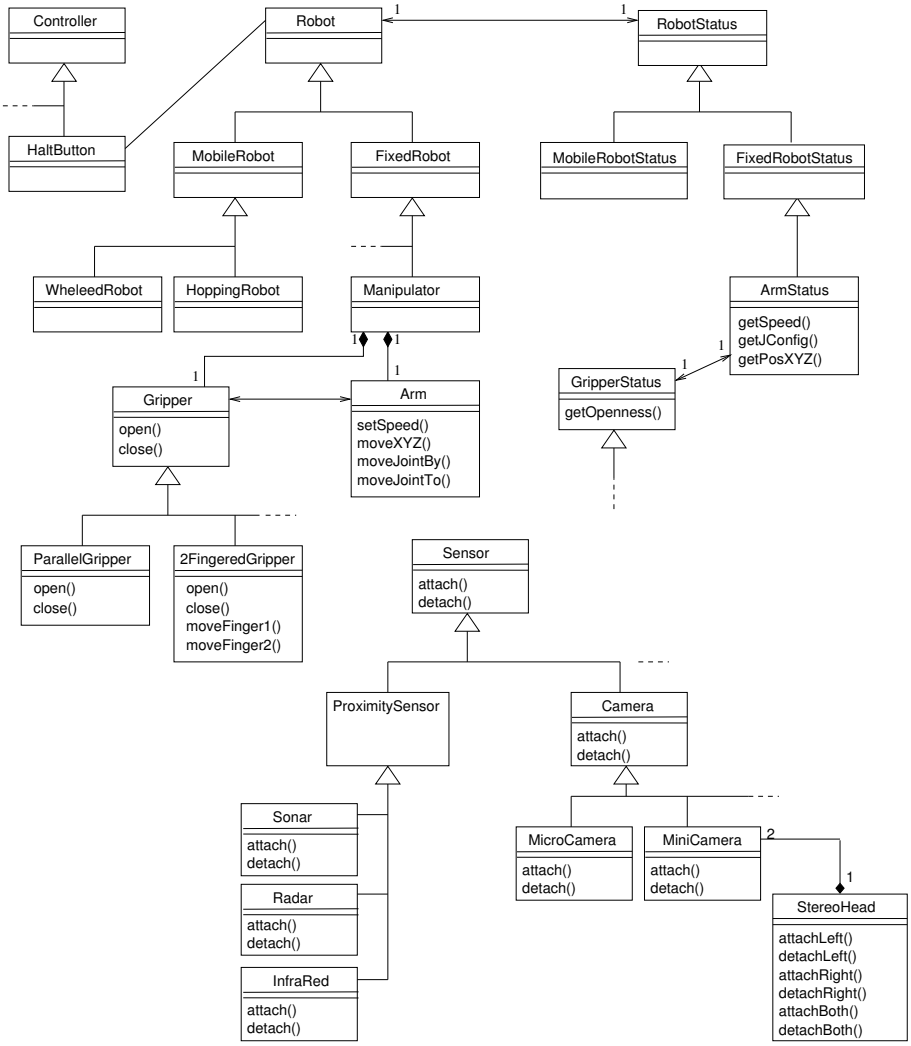

Fig. 3. Class hierarchy of the framework.

which objects are created with assigned execution priority, and `CLIENT_PROPAGATED`, in which the Client establishes the priority of the methods it invokes, and this priority is 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 safety 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 Pool avoids the overhead of thread creation/destruction at run-time and helps in guaranteeing performance by constraining the maximum number of threads at 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.

4.2 Concurrency in action execution

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. 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 requiring significant programming efforts and known to be error prone [18].

A more efficient way to perform non-blocking invocations is provided by the CORBA Messaging specification [19], through the **Asynchronous Method Invocation (AMI)** model, with either a Polling or a Callback approach. AMI allows a CORBA-based system to efficiently activate multiple concurrent actions at a remote teleoperated site. Moreover, as AMI and SMI share the same object interface, Clients can choose between synchronous or asynchronous calls while Server implementation is not affected.

In robotic applications, a set of parallel actions must often begin at the "same" time, as their coordinated execution is required to ensure logical correctness or safety. This is the rationale for the introduction of a *waiting rendezvous* strategy [20] in the framework. Proper thread synchronization in the server is required to achieve this capability. The basic CORBA synchronization mechanism is the **Mutex** interface, whereas CORBA lacks higher level mechanisms, such as condition variables, semaphores, and barriers. Hence we have developed an action synchronization mechanism on top of the Mutex. An instruction termed

`cobegin(n)` prefixes the invocation of parallel actions in a Server, acting as a barrier for the next n method invocations, whose execution, therefore, does not start until all calls have reached the server. 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.

4.3 Managing multiple Clients

A fundamental challenge in advanced teleoperated systems is to manage input from all operators while generating a single and coherent control sequence for each robot, allowing collaborative and coordinated teamwork [21].

The basic functionality provided by the framework ensures atomicity of calls to library functions devoted to the interaction with the robot controller, regardless of its thread-safeness. Potential conflicts arising from multiple Clients are avoided forcing an exclusive access to library functions through the RTCORBA::Mutex construct, implementing the mutual exclusion lock. The server side is solely responsible for the implementation of this functionality since Mutexes are introduced only in the servant code.

In addition to ensuring single command consistency and atomicity, the framework implements concurrent access control at session level, guaranteeing a full robot control without undesiderable interferences from other operators.

The implementation of the coordination scheme allowing multiple clients to control a single robot through a coherent and logically safe pattern of interaction is based on the CORBA **Concurrency Control Service** [22]. The Concurrency Service defined in the CORBA specification allows several Clients to coordinate their access to a shared resource so that the resource consistent state is not compromised when accessed by concurrent Clients. This service does not define what a resource is. It is up to the developer to define resources and identify situations where concurrent access to the resources leads to a conflict.

The coordination mechanism provided by the Concurrency Service is the lock. Each shared resource should be associated with a lock, and a Client must get the appropriate lock to access a shared resource. Several lock modes (read, write, upgrade, intention read, intention write) are defined, allowing different resolution of conflict among concurrent Clients. The specification defines two types of Client for the Concurrency Service: a transactional Client, which can acquire a lock on behalf of a transaction, and a non-transactional Client, which can acquire a lock on behalf of the current thread. Our framework adopts a non-transactional style, since most available RT CORBA implementations do not support transactional Clients yet.

In a scenario where several users compete to control a single robot and/or access data from multiple sensors, exclusive control of the robot must be granted to only one user in a given interval of time, while simultaneous read of sensor data should be allowed to other users as well. The scheme in Figure 4 shows how the framework copes with this requirement using the Concurrency Service. For each robot a Robot and a RobotStatus objects are created. The RobotStatus

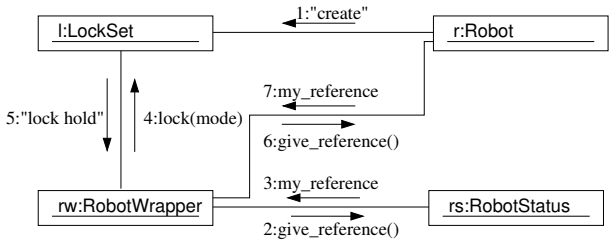

Fig. 4. Collaboration diagram describing a Client (whose core is an object of the RobotWrapper Class) that asks a lock before it is able to control a Robot object.

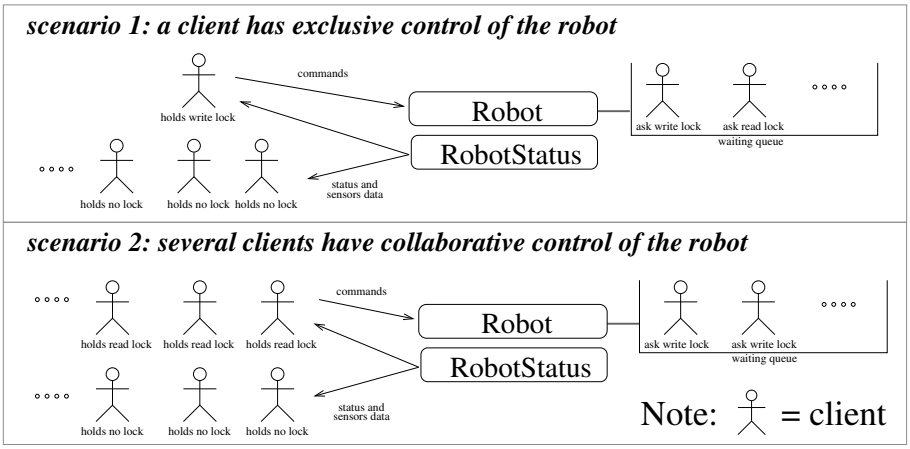

Fig. 5. Two scenarios of concurrent access to a single robot device from several Clients.

class maintains information about a robotic device while the Robot class controls movements and sets parameters. Then, for each Robot object, a CORBA:: CosConcurrencyControl:: LockSet object is created and registered in the Naming Service.

At the Client side a RobotWrapper object contains all the references to CORBA objects and interacts with the Concurrency Control Service to enforce the correct concurrent access. As shown in Figure 5 (scenario 1), the Client invoking commands on the Robot object holds a write lock ensuring exclusive control. Indeed, as the write lock conflicts with any other locks, a Client requesting a lock on the same resource will be suspended waiting its release. Clients invoking methods on the RobotStatus object, instead, are not required to hold locks as the class has only "accessor" methods.

To preserve generality and cover a wider domain of applications [21], an alternative scenario can be outlined, where a group of users want to control a single robot in a collaborative way (e.g. a "primary" operator with some "backup" operators), while preventing further operators from obtaining exclusive control of the robot. In this scenario (Figure 5, scenario 2), a collaborating Client holds a

read lock. Since the read lock conflicts only with write and intention write locks, it allows sharing of robot control with other Clients holding read locks, whereas Client requesting exclusive control through a write lock are suspended in the waiting queue.

4.4 Data distribution

Telerobotic applications often require that large amounts of sensory data be returned to remote clients. Implementing data distribution according to the Client/Server communication model has several drawbacks, such as the inactivity of the Client while waiting for a response and the saturations effects on both the network and the Server for required polling operations. A more suitable solution for interactions among peers is the *Publisher/Subscriber* communication model [23]. Whenever a Publisher (e.g. a sensor) changes state, it sends a notification to all its Subscribers. Subscribers, in turn, retrieve the changed data at their discretion.

In this section two variants of the Publisher/Subscriber communication model defined by CORBA specification are investigated, along with a Callback-based technique. In these solutions, the peers involved in the communication do not exhibit the Client/Server relationship anymore, therefore a more suitable terminology defines *Supplier* the peer producing the data, and *Consumer* anyone who receives them.

Distributed Callbacks. Our first approach for a data distribution subsystem, avoiding polling operations and minimizing network saturation, was based on Distributed Callbacks [24]. As the Observer pattern suggests, we defined two CORBA classes for each available sensor: the Subject at the Supplier side and the Observer at the Consumer side (Figure 6). To receive data from a sensor, the Consumer calls a method attach (AMI) on the remote Subject object, passing a reference to an Observer object. Each sensor holds a list of all Observer objects that have been attached. When new sensor data are ready, they are sent by the Supplier application to all the "attached" Observer objects, by invocation of the appropriate method.

Though this approach avoids Client active waiting and network saturation, the efficiency of the Supplier is greatly affected by the number of Consumers. When multiple Consumers are attached, the Supplier is supposed to persistently store their references and send a separate message to each in turn according to their preferences (each Consumer should be able to define the desired data receiving rate to avoid unnecessary data submission). Therefore, due to the high memory and computation requirements, scalability is bounded to a relatively small number of Consumers.

Event Service. To relieve the Supplier of administrative duties related to Consumers management, we implemented a second version of the data distribution subsystem based on the CORBA Event Service [25]. The general idea of the

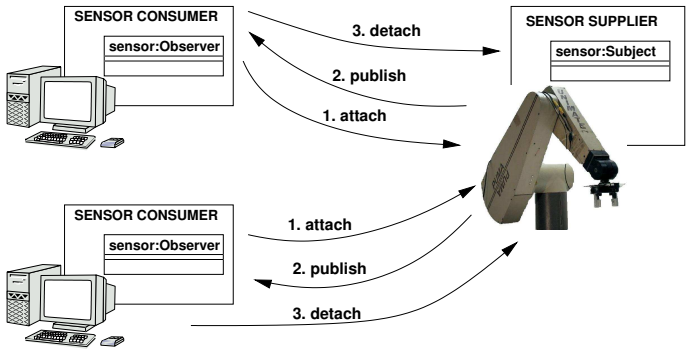

Fig. 6. Distributed callback for data distribution.

Event Service is to decouple Suppliers and Consumers using an *Event Channel* that acts as a Proxy Consumer for the real Suppliers and as a Proxy Supplier towards the real Consumers. This implementation also allows a transparent implementation of the broadcast of sensor data to multiple Consumers.

The CORBA specification proposes four different models interleaving active and passive roles of Suppliers and Consumers. For robotic applications the only reasonable model seems to be the Canonical Push Model, where an active Supplier pushes an event towards passive Consumers registered with the Event Channel.

Despite the benefits introduced by the adoption of an Event Channel, there are several matters of discussion. First of all, to avoid compile-time knowledge of the actual type of the "event", sensor data must be communicated as an Interface Definition Language (IDL) `any` type, that can contain any OMG IDL data type. The communication is therefore type-unsafe and Consumers are charged with the duty of converting the `any` type toward the data type they need. Moreover, the Event Service specification lacks event filtering features: everything is conveyed through the Event Channel, that in turn sends everything to any connected Consumer. Once again, the load of a missing property is laid on the Consumers that are forced to filter the whole data, looking for the ones they really need. Finally, the Event Service specification does not consider QoS properties related to priority, reliability, and ordering. Attempting to ensure these properties in an application results in proprietary solutions that prevent ORB interoperability.

Notification Service. Our third solution for the data distributed subsystem is based on the CORBA Notification Service [26], recently introduced in the CORBA specification to overcome the previously listed deficiencies of the Event Service.

The Notification Service is defined as a superset of the Event Service, enhancing most of its components. Notable improvements with respect to the Event Service include filtering and QoS management. Through the use of filter objects, encapsulating one or more constraints, each Consumer subscribes to the precise

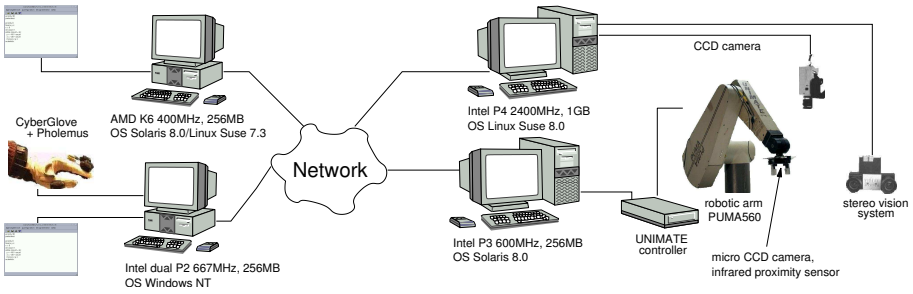

Fig. 7. The experimental testbed.

set of events it is interested in. Two filter types are defined: forwarding filter, which decides whether the event can continue toward the next component, and mapping filter, which defines event priority and lifetime. Moreover, QoS properties for reliability, priority, ordering, and timeliness can be associated to a Channel, to a Proxy, or to a single event.

5 Building a telerobotic application based on the Framework

We have developed a telerobotic application as a case study to evaluate the framework described in section 4. The application is shown in Figure 7. An operator, provided with direct continuous teleoperated control of the remote device, is required to perform a peg-in-hole task. This is a representative assembly task problem, and is often exploited for benchmarking teleoperation architectures.

The remote manipulator is an Unimation Puma 560, interfaced to a Pentium based workstation with Solaris OS running the RCI/RCCL robot programming and controlling environment. The sensory system comprises a black and white camera and an IR proximity sensor mounted near the gripper of the manipulator, a video camera mounted on the ceiling shooting the testbed area, and a stereo vision system in front of the task site. All workstations are connected in a LAN with a Fast Ethernet switch, which does not introduce substantial delay (latency is less than $100\mu s$). On the software side, the implementation is written in C++ and based on The ACE ORB (TAO) [27], a freely available, open-source, and standard-compliant real-time implementation of CORBA. Each Client station runs a Client Application allowing to graphically choose the required services among those available in the system. A more advanced station is also provided with a multimodal user interface including a virtual reality glove and a six d.o.f. tracker of the operator's wrist.

Client. The *Client Application* is built upon CORBA services providing transparency about location and implementation of the available components. The user can search for components (CORBA objects) looking for a Naming Service,

that will locate requested objects based on their name and return the reference to the remote object stored under that name. The teleoperation application includes several heterogeneous sensors whose data must be broadcasted to the Client stations and quickly returned to the user.

For definition of application tasks, the operator can choose between two alternatives: submitting a sequence of single commands, or developing a program in a C-like high level language that will then be interpreted. To support invocation of concurrent actions at the server and achieve temporal continuity in command execution by the robot, the developed application takes advantage of asynchronous calls (see section 4.2).

Server. The *Server Application* manages the Puma manipulator and several sensors according to Client requests, whose order and urgency should be preserved. The server has been implemented as a multi-threaded architecture using the Thread Pool mechanism (see section 4.1). This feature allows management of the high number of requests that could be received from multiple Clients concurrently monitoring and controlling the task.

For safety reason an "Emergency Button" object is implemented in the Server to immediately stop the system in emergency situations. To this purpose, the Thread Pool with Lanes mechanism preserves server reactivity to high priority calls, avoiding priority inversion and thread exhaustion by low priority requests.

6 Empirical performance assessment

Several variants of the application 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 in the Server application that controls the manipulator and the sensors which are directly related to the manipulator. 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 is a Priority Level Round Robin (`SCHED_RR`), which is available in any POSIX-compliant operating system.

Many experiments involving simulated workload have been carried out to evaluate the correctness and robustness of the Server, which has been tested with a variety of sets of concurrent actions, with different priority levels and synchronization. A goal of these experiments was to verify the effectiveness of `cobegin(n)` to avoid priority inversion in the execution of parallel actions. One of the experiments is described in Figure 8 (left), 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 8 (right) 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

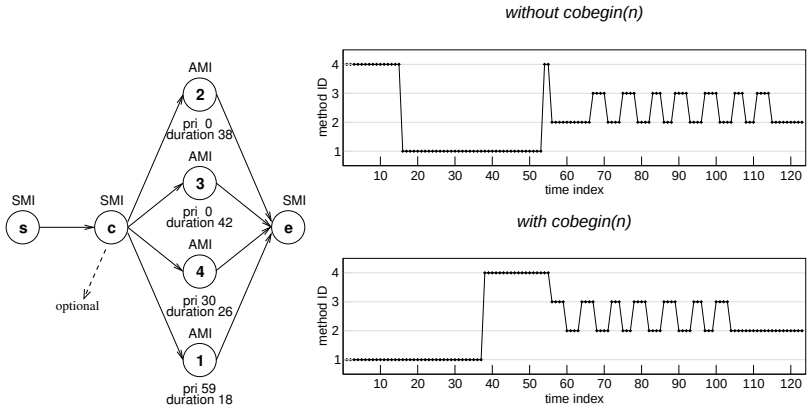

Fig. 8. (left) Precedence graph of a concurrent task, consisting of an initial action `s=start()`, an optional `c=cobegin()`, followed by four concurrent actions with different priority and duration. (right) Experimental results of concurrent actions without (top) and with (bottom) `cobegin(n)`.

Host name	Hardware configuration	Operating system
Trovatore (faster machine)	PIV 2.4GHz, 512MB RAM	SuSE Linux 8
Malcom (slower machine)	Athlon 800MHz, 256MB RAM	Mandrake Linux 9.1

Table 1. Experimental Setup: Host features.

(ID 1). With `cobegin(n)` (bottom diagram), the priority of threads is always guaranteed and no priority inversion occurs.

6.1 Data Distribution

The communication models described in Section 4.4 have been integrated in the proposed CORBA-based framework for telerobotic systems. The experiments reported in the following only assess the relative performance in terms of latency and scalability of the three proposed data distribution mechanisms. All experiments reported in this section follow the Push Model: a Supplier generates data and sends them to the Event Channel, when available, or directly to Consumer processes. A single Supplier and one ore more Consumers, all requiring the same sensory data, are considered. Both Supplier and Consumer(s) are located on the same host, whereas the Event Channel can be on a different host.

Two host machines exploited in the experiments are listed in Table 1 along with their features. The hosts are connected via a Fast Ethernet switch. Aside from our tests, the network had no significant traffic nor was any other processing taking place on these hosts.

In the first set of experiments a 64 Byte packet is pushed by the Supplier to a single Consumer. Consumer activity is limited to the update of the latency

Implem.	EC,S,C on faster machine t_{min}	t_{avg}	$jitter$	EC on slower machine S,C on faster machine t_{min}	t_{avg}	$jitter$	EC on faster machine, S,C on slower machine t_{min}	t_{avg}	$jitter$
Distr. Call.	0.12	0.38	0.58	0.11	0.14	0.11	0.23	0.30	0.44
Event Serv.	0.33	0.62	0.54	0.42	1.05	0.33	0.53	1.17	0.70
Notif. Serv.	0.40	0.72	0.58	0.50	1.41	0.20	0.61	1.26	0.26

Table 2. Latency (ms) with a 64 Byte packet with three configurations of Supplier(S), Consumer (C), and Event Channel (EC).

Implem.	EC,S,C on faster machine N=1	N=3	N=10	N=50	N=70	N=100	EC on faster, S,C on slower machine N=1	N=3	N=10	N=50	N=70	N=100
Distr. Call.	0.73	0.98	3.76	37.76	45.53	82.77	0.66	2.23	7.64	51.90	85.90	144.1
Event Serv.	1.25	1.48	4.52	38.00	41.94	71.46	1.68	2.96	7.11	33.59	49.42	77.6
Notif. Serv.	1.38	1.67	5.15	40.60	45.18	76.34	1.82	3.23	7.75	36.47	53.09	84.32

Table 3. Average interarrival time (ms) with a 64 Byte packet, increasing the number of Consumers (N) for two configurations of Supplier (S), Consumers (C), and Event Channel (EC).

value so far. Table 2 reports latency minimum, average, and standard deviation (jitter) values on a set of 50,000 samples considering alternative allocations of Event Channel, Supplier, and Consumer. Allocation of the Event Channel only affects implementations based on Event or Notification Services. Of course, due to the single Consumer type of experiment, the Distributed Callback approach exhibits a lower latency than Event and Notification Services implementations (whose additional features are not utilized). The added latency of Event and Notification Services is small when the Event Channel is located in the same host as the Supplier and Consumer. When the Channel is located on another machine the performance of CORBA services decreases, since data are sent forth and back to the Channel host through the network.

The next set of experiments measures the average time interval between two successive 64 Byte packet receptions (interarrival time) increasing the number of Consumers from 1 to 100. Results (in ms) for selected Consumer configurations are reported in Table 3. At the beginning of the range investigated, the Callback implementation has slightly better performance than Event Channel-based ones, because it requires data to cross a lower number of hops. However, Event Service and Notification Service implementations have better scalability when the Event Channel is located on a fast machine: they achieve a performance comparable to the Callback implementation starting from 10 Consumers, and significantly better performance starting from 70 Consumers.

To summarize, for robot Servers performing several complex tasks (e.g., sensor data acquisition and distribution, motion planning, actuator control) and dealing with a large number of attached Clients, the Event Channel represents an effective tool to maintain the overall workload under control. When Clients

have different QoS needs and at the expense of a slight overhead, the Notification Service is the most appropriate solution, thanks to its configurability options not available in Callback and Event Service implementations.

7 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 [14]. 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* [11], 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 [28], a supervisory control scheme [29], or an Internet telerobotic system conceived to provide assistance to aged and disabled people [30].

Two recent papers are more directly concerned with the implementation of systems supporting distributed telerobotic applications. Hirukawa and Hara [9] propose a framework based on OO programming for robot control, whereas Dalton and Taylor [11] 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 [11] exploited non-standard middleware. We believe that with the current CORBA and RT CORBA specifications, including the AMI invocation model and other advanced features, 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 framework 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 [11]. Now that RT CORBA technology has matured, it can be leveraged upon to develop reliable COTS-based telerobotic systems with strict control over computational resources.

8 Conclusions and future work

The viability and cost effectiveness of new telerobotic applications such as virtual laboratories, networked and on-line robots can be widely enhanced exploiting COTS-based software components. Moreover, systems implementing those applications pose also demanding challenges: they should be dynamically reconfigurable and highly scalable to deal with a potentially large number of peers, they should provide real-time features, guaranteed performance and efficient concurrency mechanisms, both locally and in a distributed environment. CORBA

middleware, especially with its recent extensions, seems well suited for the needs of distributed telerobotic systems. In this paper we have summarized our experience in the development of a software framework for telerobotics based on Real-Time CORBA, ensuring proper real-time operation of the server and managing concurrent control and data distribution with multiple Clients.

The results obtained show that CORBA brings a number of remarkable advantages in the telerobotic domain, enabling portable, highly reconfigurable applications with support for concurrency and real-time features. Furthermore, CORBA standard services for naming resolution, data distribution, and concurrency control avoid the need for ad-hoc solutions, which are often error-prone and require significant development efforts.

The major drawbacks encountered in our experience are some overhead in communications, the limited synchronization mechanims available, and, more important, the fact that none of the CORBA ORBs available offers a full implementation of the CORBA standard, i.e. covering aspects such as dynamic scheduling, fault-tolerance, fully compliant CORBA services.

We now plan to investigate additional techniques for distributing data with minimal overhead, for managing authentication of operators and for secure access to the telerobotic system. Further tests and software design efforts are also required for public distribution of the framework.

Acknowledgment

This research is partially supported by MIUR (Italian Ministry of Education, University and Research) under project RoboCare (A Multi-Agent System with Intelligent Fixed and Mobile Robotic Components).

References

1. C. Zielinski, "Object-Oriented Robot Programming," *Robotica*, vol. 15, no. 1, Jan. 1997.
2. T. Hori, H. Hirukawa, T. Suehiro, and S. Hirai, "Networked Robots as Distributed Objects," in *IEEE/ASME Int. Conf. on Advanced Intelligent Mechatronics*, 1999.
3. S. Bottazzi, S. Caselli, M. Reggiani, and M. Amoretti, "A Software Framework based on Real-Time CORBA for Telerobotic Systems," in *IEEE Int'l Conf. Intelligent Robots and Systems*, 2002.
4. S. Jia, Y. Hada, Y. Gang, and K. Takase, "Distributed Telecare Robotic Systems Using CORBA as a Communication Architecture," in *IEEE Int. Conf. Robotics and Automation*, 2002.
5. T. Ortmaier, D. Reintsema, U. Seibold, U. Hagn, and G. Hirzinger, "The DLR Minimally Invasive Robotics Surgery Scenario," in *Work. Advances in Interactive Multimodal Telepresence Systems*, 2001.
6. C. Preusche, J. Hoogen, D. Reintsema, G. Schmidt, and G. Hirzinger, "Flexible Multimodal Telepresent Assembly using a Generic Interconnection Framework," in *IEEE Int. Conf. Robotics and Automation*, 2002.
7. C. Sayers, *Remote Control Robotics*. Springer, 1999.

8. T. B. Sheridan, *Telerobotics, Automation, and Human Supervisory control*. Cambridge, MA: MIT Press, 1992.
9. H. Hirukawa and I. Hara, "Web-Top Robotics," *IEEE Robotics & Automation Magazine*, vol. 7, no. 2, pp. 40–45, June 2000.
10. K. Goldberg, S. Gentner, C. Sutter, and J. Wiegley, "The Mercury Project: A Feasibility Study for Internet Robots," *IEEE Robotics & Automation Magazine*, vol. 7, no. 1, pp. 35–39, Mar. 2000.
11. B. Dalton and K. Taylor, "Distributed Robotics over the Internet," *IEEE Robotics & Automation Magazine*, vol. 7, no. 2, pp. 22–27, June 2000.
12. H. Hirukawa, I. Hara, and T. Hori, "Online robots," in *Beyond Webcams: an introduction to online robots*, K. Goldberg and R. Siegwart, Eds. The MIT Press, 2001.
13. *WS2001: International Workshop on Tele-Education in Mechatronics Based on Virtual Laboratories*, Weingarten, Germany, July 2001.
14. K. Goldberg and R. Siegwart, Eds., *Beyond Webcams: an Introduction to Online Robots*. The MIT Press, 2001.
15. P. Backes, K. Tso, and G. Tharp, "Mars Pathfinder Mission Internet-Based Operations using WITS," in *IEEE Int. Conf. Robotics and Automation*, 1998.
16. Y. Tsumaki, T. Goshozono, K. Abe, M. Uchiyama, R. Koeppe, and G. Hirzinger, "Verification of an Advanced Space Teleoperation System using Internet," in *IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems*, 2000.
17. *Real-Time CORBA Revision 1.1*, Object Management Group, Aug. 2002.
18. A. Arulanthu, C. O'Ryan, D. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proc. of the Middleware 2001 Conference ACM/IFIP*, 2000.
19. *The Common Object Request Broker: Architecture and Specification Revision 3.0*, Object Management Group, Dec. 2002.
20. B. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.
21. N. Chong, T.Kotoku, K. Ohba, K. Komoriya, N. Matsuhira, and K.Tanie, "Remote coordinated controls in multiple telerobot cooperation," in *IEEE Internatial Conference on Robotics and Automation*, 2000.
22. Object Management Group, "Concurrency Service Specification," http://www.omg.org/technology/documents/formal/concurrency_service.htm, Apr. 2000.
23. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns*. Wiley and Sons, 1996.
24. M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.
25. Object Management Group, "Event service specification, v. 1.1," http://www.omg.org/technology/documents/formal/event_service.htm, Mar. 2001.
26. ——, "Notification service specification, v. 1.0.1," http://www.omg.org/technology/documents/formal/notification_service.htm, Aug. 2002.
27. Distributed Object Computing (DOC) Group, "Real-time CORBA with TAO (The ACE ORB)," http://www.ece.uci.edu/~schmidt/TAO.html.
28. C. Paolini and M. Vuskovic, "Integration of a Robotics Laboratory using CORBA," in *IEEE Int. Conf. Systems, Man, and Cybernetics*, 1997.
29. R. Burchard and J. Feddema, "Generic Robotic and Motion Control API Based on GISC-Kit Technology and CORBA Communications," in *IEEE International Conference on Robotics and Automation*, 1997.
30. S. Jia and K. Takase, "An Internet Robotic System based Common Object Request Broker Architecture," in *IEEE Int. Conf. Robotics and Automation*, 2001.