# Advanced Teleoperation Architecture

Andrea Castellani, Stefano Galvan, Debora Botturi, and Paolo Fiorini

Department of Computer Science University of Verona, Italy
{castellani, galvan, botturi, fiorini}@metropolis.sci.univr.it

## 1 Introduction

In this Chapter we report on the efforts carried out at the Robotics Laboratory ALTAIR of the University of Verona (Italy) towards the development of a high performance architecture for bilateral, i.e. force reflecting, teleoperation system. Key issues for the advancement of teleoperation technology include high performance control dealing with problems such as unpredictable delays and low bandwidth, advanced mechanisms and quantitative measures on the complete system including haptic displays. In the area of software development, new methodologies have been developed that permit a very careful and precise design, and allow to implement high performance architectures at a reasonable effort in terms of cost and development time. Our architecture, called Penelope, takes into account very important features of a haptics system: real-time behavior, distributed resources and general purpose structure. However, we are also studying features that make it particularly suited for high reliability and safety operations, such as space teleoperation and robot-assisted surgery.

In summary, the main features provided by the architecture proposed here, are:

- Communication between elements implemented with ACE CORBA.
- Applications written in C++ under the RTAI Linux Operating system.
- CORBA class allowing object communication using the C++ CORBA IDL interface.
- Inheritance from CORBA class for each "real" class.
- Communication independent of the real location of the implemented class and of the communication mechanism.
- Methods called by procedures without knowledge of their physical location within the teleoperation system.
- Security and robustness ensured by UML based design, enforcing software specifications.

D. Brugali (Ed.): Software Eng. for Experimental Robotics. STAR 30, pp. 409–430, 2007.

This architecture is under test with the ALTAIR setup in force feedback demonstrations. The teleoperation system consists of a PUMA 200 and a PUMA 560, two NASA-JPL Force Reflecting Hand Controllers (FRHC) with force and torque reflection capability in any arbitrary direction and orientation, and force sensors mounted on the robot wrists.

In the following, we present a our design choices, we give their motivations, and describe the laboratory setup that we are using to test the various features. Emphasis is given on the performance expected from a software architecture developed for a teleoperation purpose.

## 2 Teleoperation

Teleoperation is defined as the control over a distance of one or more devices by a human operator. In this context, we assume that the controlled devices are robots. Usually teleoperation refers to a system with a master/slave configuration, where the operator works on a master joystick that is kinematically compatible with the slave manipulator (see Figure 1).

It has been shown that operator performance is improved by providing force information to the human operator [DZK92]. Force information can be presented visually to the operator on a monitor, but the most significant performance improvement is achieved by providing force feedback to the operator, i.e. by generating forces directly with the motors of the master device. In this case the operator is said to be kinesthetically coupled to the slave and the teleoperator system is said to have bilateral control or to be force reflecting. Haptic feedback devices were pioneered in teleoperation systems as far back as the 1940's. In teleoperation applications of haptics, a loop is closed between the human operator's motion "input" and forces generated by the haptic device based on data received via a communication link from the robot manipulator and the environment. Teleoperation is used in cases where the environment is not directly accessible by a human, or when it is too dangerous for an operator, or when it is necessary to scale the force exerted on the environment. Almost all of the applications of teleoperation involve contact with the remote environment, e.g. grasping, welding and puncturing. In particular, robotic applications to surgery include puncturing as one of the most usual actions. The trade-off between stability and performance is the main consideration in the design of a teleoperation system. Mainly, the stability problems are due to the drawbacks of the communication over a distance, *unpredicatable time delays* and *low bandwidth*. Therefore, those problems reduce perception of environment, making uncomfortable and unsafe to work with the system. Teleoperation control architectures analyzed in the literature can be classified in terms of their stability and performance trade-off's [Law93]. Control algorithms for ideal kinesthetic coupling [YY94] are at one end of the spectrum, whereas passivity based algorithms [AS92] are at the other end.

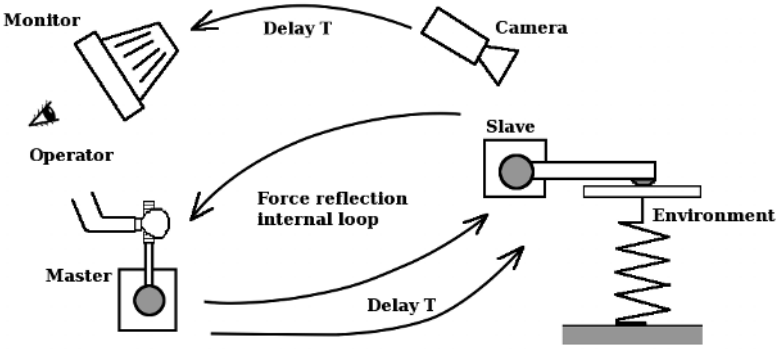


**Fig. 1.** Teleoperation schema.

Conventional algorithms such as position error based force feedback and kinesthetic force feedback lie in the middle. Both performance and stability are inherently dependent on the task for which the system is designed. Thus the need to design system controllers for the specific applications, including local and remote environments, not only in terms of parameter tuning, but also with respect to the overall control scheme.

In a teloperation environment, and especially for research purposes, we want to have the possibility to easily interconnect different devices, test different control algorithms, process data from several and different sensors and use different connection media. From this assumption, the idea of a middleware integration framework arises. The framework has to be designed to enhance the ability of modularize, extend and reuse the software infrastructure in order to handle a distributed environment. In addition some real-time capability is needed to maintain transparency and stability while data are sent over the communication channel, dealing with unpredictable time delays and low bandwidth. Moreover, the main advantages of distributed real-time middleware for teleoperation are reduced systems cost, remote expertise on demand and dynamic access to the system [She92a]. One possible approach is the so-called whitebox framework [GHJV95]. This framework relies on the object-oriented nature of the programming language used (inheritance, abstraction, dynamic binding) to achieve modularity, reusability and extensibility properties. Whitebox frameworks require a good knowledge of the complete architecture in order to create a suitable inheritance hierarchy. A top-down approach is required, and to reduce complexity and to ensure robustness of the software a UML schema can be provided.

The Unified Modeling Language (UML) [Gro] is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. There are many different UML diagrams to cover different software engineer-

ing problems that are described in papers and/or books [Ric99]. With respect to this project, we have used only the *Class Diagram*. We have used the UML Class Diagram firstly in specification perspective to understand which classes were necessary and what should be their relationships. During implementation, we used the implementation perspective to organize the complex software structure and to test its operation.

In distributed telerobotic applications, systems are extremely heterogeneous, they are often implemented using previously available devices, based on specialized hardware, running on different operating systems, and programmed in many different languages. Therefore, middleware software for communication and for distributed object computing is a necessity to guarantee the overall system functionality. Available solutions to resolve the heterogeneity problem are: Microsoft Distributed Component Object Model (DCOM), Java Remote Method Invocation (Java RMI) and Common Object Request Broker Architecture (CORBA). In order to have general-purpose architecture and language, vendor, and operating system independence, CORBA is the solution adopted in the research described here.

CORBA is an emerging open distributed object computing infrastructure being standardized by OMG [OMG]. The basic CORBA paradigm is that of a request for services of a distributed object. The services that an object provides are given by its interface. Interfaces are defined in OMG's Interface Definition Language (IDL). Distributed objects are identified by object references, which are typed by IDL interfaces. More about CORBA can be found in [OMG] and about TAO Real Time CORBA in [HV99].

Deterministic performance is achieved using standard Linux drivers, applications and functions, in the RTAI (Real Time Application Interface) [DIA] of the standard Linux kernel, providing the ability to make it fully pre-emptable. RTAI [DIA] extends the standard (vanilla) kernel by providing different schedulers that permit to give the highest priority to a specific time-critical process, delaying interrupts from the Linux operating system. The real-time code has to be written in the form of a Linux kernel module. This permits its full integration with the system and the possibility to define the new scheduler and activate/deactivate preemption. Several communication functions are provided to permit exchange of information between the kernel "realm" and standard user space programs. All traditional IPC (Inter-process Communication) functions are provided such as semaphores, mailboxes, FIFO's and shared memory. Dynamic Memory Allocation (DMA) and memory lock in hard real-time tasks are also available.

Another enhancement to the aim of deterministic performance can be achieved moving the bottlenecks present in the control loops of the devices to hardware. This can be done by using specific interfaces to fulfill the middleware framework requirements and preserving aspects such as transparency and software independency. Using CORBA, other important aspects such as security and fault tolerance are ensured by using distributed and redundant resource management. Local control and the network are decoupled guaran-

teeing system robustness. This approach helps in the design of new modules by providing a common interface structure to match the architecture requirements and, by using this architecture during implementation a more conscious software evolution can be acheived. Another advantage is that developers do not have to deal with the communication layer and thus they can reuse the code more easily. However, the top-down approach is very difficult to follow during the software development phase. In same cases in fact, it is necessary to modify the software specifications and to change some design characteristics because new needs have arisen during implementation. Therefore, another goal of the middleware development should be to make the modification process easier.

## 3 Penelope ($\Pi\eta\nu\epsilon\lambda o\pi\epsilon\iota\alpha$)

Penelope is a modular and distributed software architeture, aimed specifically at teleoperation systems. With this architecture, it is possible to control, in a distributed framework, several types of robots in a simple way and test different teleoperation configurations, including new devices, sensors and boards (e.g. I/O boards, FPGA boards and Multi-Axes Controller boards). Initially, to develop this type of structural software we have analyzed how to implement the structure and which tools or technologies to use. Software for teleoperation must be modular and easy to expand or modify. In a robotic laboratory there are different robots controlled by different devices or boards that can be used together with different sensors. For these reasons, we have chosen to implement Penelope using Object Oriented Programming (OOP). In this way, every real object in the teleoperation framework is represented by a specific object in the software. With this programming choice, it is also possible to use hierarchy, polimorfism, function overloading and all the other OOP mechanisms to develop a complex and modular software system.

In a general teleoperation system, it is possible (and appreciated) that objects (robots (master or slave), sensors, graphic interfaces) are used, controlled or connected to different PC stations. In some cases, different Operating System are used and therefore a communication protocol or, in general, a communication infrastructure is necessary. A possible approach is to develop a dedicated system to ensure master/slave (or other device) communication [She92b]. However, this solution, which is the natural choice in a critical teleoperation architecture, is economically unacceptable in most cases. To obtain a general and more powerful communication infrastructure, we have chosen to use CORBA. With this communication framework, it is possible to distribute Penelope over different PC stations simply by using the object interconnection service of CORBA (OOP helps in using CORBA in the code). Communication is implemented using the real-time implementation of CORBA, The ACE ORB middleware (TAO), which allows to control the communication timing

more precisely, and to use different communication protocols or scheduling priorities, when necessary.

Penelope is an architecture developed with research objectives in a research environment. For these reasons, we have implemented all the infrastructure in a free Operating System using Linux under a GPL (General Public License). To respect strict timing constraints, for example in control algorithms, we have used the RTAI Linux extension presented in the previous Section.

Given the structure of Penelope, the three most important features that must be described in detail are:

- The architecture skeleton (structure of the classes);
- Object communication management when they are located on different PC's;
- The robot control to satisfy real-time constraints.

Another important feature of Penelope is its Graphical Interface independence. Penelope can be used by different types of Graphical User Interfaces (GUI), by means of different user interface builders and different graphical libraries. GUI can run and communicate with Penelope on various OS and also using a web interface. Therefore, a GUI is not a key feature of this architecture and it will not be described here. Each user can implement his own personal interface and use Penelope only knowing the IDL class interfaces.

After describing Penelope main features, we will present an example of a master/slave teleoperation system and we will discuss early performance results in communication and control tasks.

### 3.1 Class Structure of Penelope

A teleoperation system, and Penelope in our case, must use many different object types, for example robots controlled by different algorithms through a variety of Input-Output boards. Moreover, in many cases master/slave systems use external devices, such as visual and audio displays, or force sensors. Because of the complexity of this structure, we have designed the architecture of Penelope using the UML Class Diagram shown in Figure 2.

As shown in the Figure, Penelope main classes are `Robot`, `Board`, `Sensor`, `Controller`, and `Trajectory` that are briefly described in the following[1].

- **BoardImpl**. Represents all devices used as interface from a real robot to a PC station, e.g. to read encoders or write voltage/torque to the motors. This represents a general class from which the specific boards inherit their interface.
- **RobotImpl**. Manages all types of robots. A manipulator, a joystick, a mobile robot or a multiple robot (for example a mobile robot platform connected physically with a manipulator) are sons of RobotImpl class via

[1] The "Impl" suffix after a class name is used to differentiate real implemented classes from CORBA interface classes.

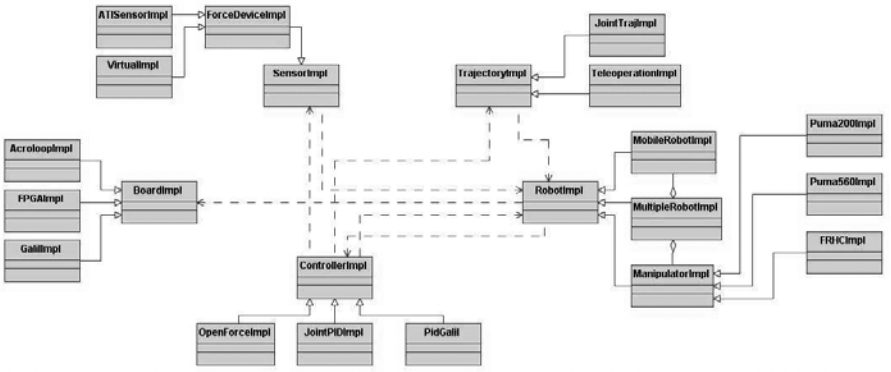


**Fig. 2.** Penelope UML Class Diagram. It represents the diagram at the specification perspective and does not consider the interface classes for communication.

inheritance. In the same way, a specific manipulator inherits from ManipulatorImpl class.

- **ControllerImpl**. This class contains all the feature necessary to control a robot; for example controller algorithms and controller parameters. As robot and board, this represents only a general class; specific types of controllers and their implementations inherit from it.
- **SensorImpl**. Interface class to different sensor devices. With inheritance it contains the force sensor class, the sonar and laser device class, and the camera device as a visual sensor.
- **TrajectoryImpl**. It produces all references followed by a robot, for this reason this class can be used in two different way:
  - as a trajectory generator (of different types) for a single robot in a simple robotic system (not in a teleoperation system);
  - as a communication channel between master and slave using, if necessary, all the sensors present in the teleoperation system. In this case, for example, TrajectoryImpl can manage the master/slave data transfer invoking the correct methods.

Clearly, there are complex relations between these classes, as shown in Figure 2. For example an instance of ControllerImpl (a ControllerImpl object) can be in relation with some instance of other objects. The control algorithm manages a robot and there must be a relation between this two types of objects. Again, a control algorithm can use a sensor, for example an instance of the force sensor in the case of a force control algorithm.

In all cases, these relations can be translated into code by means of a class attribute (a pointer to the specified class) as shown in the following example.

```
#include "trajectory.h"
#include "sensor.h"
#include "robot.h"
```

```
class controllerImpl
{
protected:
   robotImpl *robot;
   trajectoryImpl *trajectory;
   sensorImpl *sensor;

// Other attribute ...

public:
   controllerImpl();
   ~controllerImpl();
   int setRobot(int robotNumber);
   int setTrajectory(int trajectoryNumber);
   int setSensor(int sensorNumber);

// Other methods ...
};
```

As it will be explained in the next Section on communication, methods to set object relations can not use pointers and then input are codes that represent the specified object type. To understand the complexity of the Penelope objects, we briefly describe the interface of an instance of ManipulatorImpl, Puma200Impl, shown in Figure 3 and in the followinf code fragment.

```
class puma200Impl:public virtual manipulatorImpl
{
// Some Attributes

public:
   puma200Impl();
   ~puma200Impl();

public:
   int getJointPosition(float *joint_position);
   int getJointVelocity(float joint_velocity);
   int getJointAcceleration(float *joint_acceleration);
   int getJointMovements(float *position,float *velocity);
   int WSPosition(const float *angles,float *work_space_position);
   int WSPositionOther(float *work_space_position);
   int WSForce(float *fortor);
   int WSVelocity(float *work_space_velocity);
   int WSAcceleration(float *work_space_acceleration);
   int computeJointPosition(const float *work_space_position,
                            float *angles);
   int dynamic(float *vect);
   int jacobian(const float *angles,float jaco[][]);
   int transposeJacobian(const float *angles,float jaco[][]);
   int inverseJacobian(const float *angles,float jaco[][]);
   int jointTorque(const float *angles,const float *force,
                                       float *torque);
   int directKinematic(const float *angles,float *position,
                       float *versor1,float *versor2,
                                      float *versor3);
   int inverseKinematic(const float *position,const float *versor1,
                        const float *versor2,const float *versor3,
                                             float angles[]);
   int step_rad(const float *step,float *rad);
   int rad_step(const float *rad,float *step);
   int goNested();
   int goStart();
   int goWork();
```

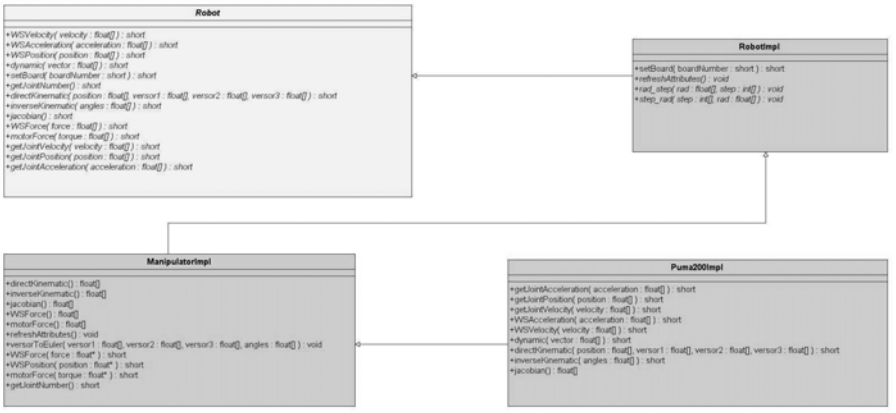


**Fig. 3.** Particular of Penelope class Diagram with RobotImpl inheritance structure and Puma200Impl interface (attributes and methods).

Some of the methods present in this class interface use the relations with other object. For example when `goWork()` is called, the PUMA 200 manipulator goes to its working position. To do this, methods use a controller object and a board object. The first to start the joint control algorithm and the second to communicate data and receive robot information. It is also possible that a single board object is connected to more than one robot, or a single robot object connected to multiple boards (for example three robot joints connected to one board and three other robot joints connected to another board). Penelope has this versatility also respect to robot control algorithms: a set of joints can be controlled with a joint-based algorithm and the remaining joints can be controlled with a different algorithm.

Another important class in Penelope is represented by TrajectoryImpl. As mentioned above, it can be used in two different ways. In general, when an object calls the TrajectoryImpl method "`getReferiment(float time,float *ref)`" it returns the current reference followed by the control algorithm in that control cycle:

- as a trajectory generator, it generates different types of trajectory in joint-space or in operational space; step, ramp, velocity or position profiles with cubic splines.
- in teleoperation, it sets the parameters to create the reference connection for controller. When the connection is ready, `getReferiment` is able to send references to controllers that can be very different. It can send joint positions if the master controls the slave movements by sending position in configuration space, or $3D$ positions if the control is in operational space, or by sending forces measured by the sensors at the slave or the master sides. Is also possible to send hybrid references as a mixture of different

data types (for example, the master sends $3D$ positions and force data measured to the slave controlled with a hybrid approach).

### 3.2 Object Communication

Penelope represents a distributed teleoperation architecture where objects can be on different PC's. Thus a communication infrastructure is needed. As mentioned earlier, CORBA TAO middleware as been used for this purpose. Any main class in the architecture inherits from the corresponding class (interface) in IDL, as shown in Figure 4. Therefore, for example, RobotImpl inherits from Robot IDL interface. All interfaces of methods in IDL are implemented in the corresponding real *C++* class and can be invoked in a distributed way. The following code shows this concept as an example of ControllerImpl where `Controller` is the interface name in IDL, ROB is an IDL module corresponding to a *C++ NameSpace* and `CORBA::Short` represents the mapping of CORBA for basic type `int`.

```
class controllerImpl:public virtual POA_ROB::Controller
{
// Attribute ...

public:
   controllerImpl();   // Constructor
   ~controllerImpl();  // Destructor
   CORBA::Short setRobot(CORBA::Short robotNumber)
                     throw(CORBA::SystemException);
   CORBA::Short setTrajectory(CORBA::Short trajectoryNumber)
                              throw(CORBA::SystemException);
   CORBA::Short setSensor(CORBA::Short sensorNumber)
                        throw(CORBA::SystemException);

// Other methods ...
};
```

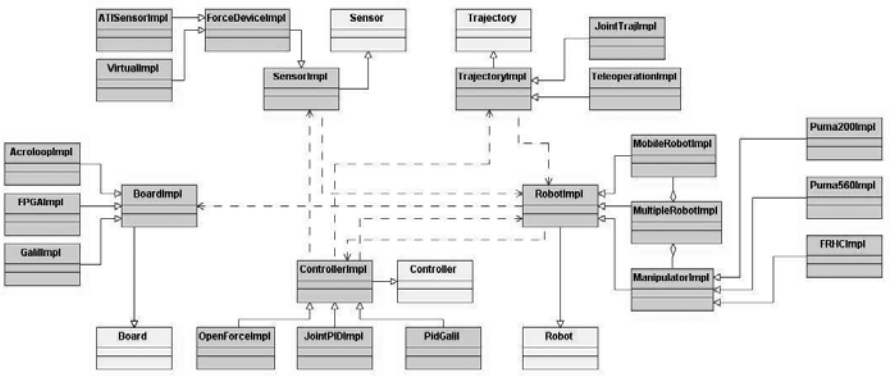


**Fig. 4.** Penelope UML Class Diagram with IDL interface for object communication.

To invoke the distributed methods of a specific object, the architecture must know their references. To do this, CORBA defines the Naming Service. Penelope uses this service to read and write references from a complex tree managed by the Naming Service. The tree structure is divided into contexts to maintain a logical representation of the architecture hierarchy as shown in Figure 5.

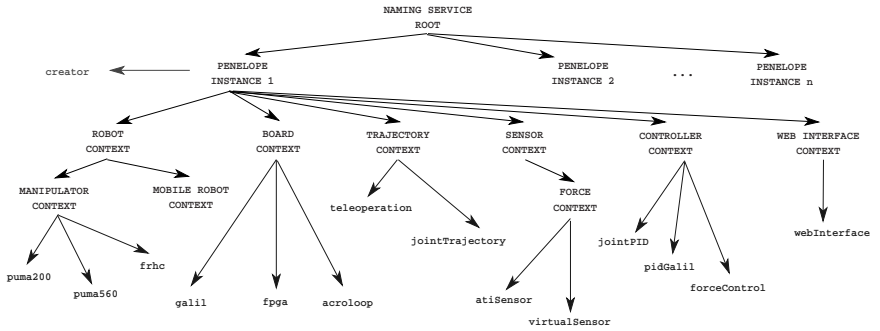


**Fig. 5.** The logical structure of Penelope CORBA Naming Service.

By means of this service, it is possible to receive any object reference and use this as a distributed object. However, a teleoperation system is more complex and Penelope must be able to create or destroy an object when demanded. When the architecture starts, no object is created and there are not references in the Naming Service. Only after the operator sets the objects the references are created. To allow this feature, Penelope uses an important class (inherited from the corresponding IDL class) named creatorImpl shown in Figure 6. When Penelope starts, only this object is active and its reference is in the Naming Service. Methods of creatorImpl can be invoked on demand, to create or destroy any other architecture object, as shown in the code fragment below, representing the interface of the creator class.

```
using namespace CORBA;
class creatorImpl:public virtual POA_ROB::Creator
{
public:
  char* rootName;                        // Name of Naming Service Root
  vector<std::string> robotNames;        // Vector Names of Robots
  vector<std::string> boardNames;        // Vector Names of Boards
  vector<std::string> controllerNames;   // Vector Names of Controllers
  vector<std::string> trajectoryNames;   // Vector Names of Trajectories
  vector<std::string> sensorNames;       // Vector Names of Sensors

  vector<int> robotID;       // Vector ID of Robots
  vector<int> boardID;       // Vector ID of Boards
  vector<int> controllerID;  // Vector ID of Controllers
  vector<int> trajectoryID;  // Vector ID of Trajectories
  vector<int> sensorID;      // Vector ID of Sensors
```

```
  vector<robotImpl *> robotVect;          // Robot pointers
  vector<boardImpl *> boardVect;          // Board pointers
  vector<controllerImpl *> controllerVect;// Controller pointers
  vector<trajectoryImpl *> trajectoryVect;// Trajectory pointers
  vector<sensorImpl *> sensorVect;        // Sensor pointers

public:
  Short objectCreator(const Short ref,const char *name)
                                  throw(SystemException);
  Short objectDestroyer(const char *name) throw(SystemException);
};
```

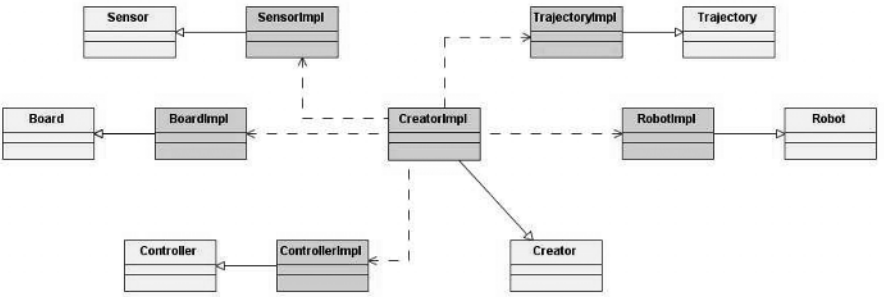


**Fig. 6.** UML Class Diagram of Penelope with Creator dependencies.

To create an object, `objectCreator()` must be invoked (using the root name and the creator reference in the Naming Service). This stores name, ID and the pointers to the created objects in a structure, and a reference with the specified name is created in the Naming Service under the correct context. When `objectDestroyer()` is invoked, the specified object with its name, ID and pointers is destroyed (deallocating occupied object memory) and the reference in Naming Service is removed. In other words, with the `objectCreator()` it is possible to manage Penelope dynamically creating the necessary distributed object at runtime, or destroying then on demand.

### 3.3 Robot Control

Some features of a teleoperation system have real-time properties. The more interesting example of this is well represented by the Robot class. A robot must be controlled and typically a control algorithm is executed on a Board and not on a PC station, because of time constraints. However, in a flexible teleoperation system such as Penelope, it must be possible to use different, and often new control algorithms. For this reason, board-based control is not the best choice. In Penelope control algorithms run on a Linux PC equipped with RTAI to satisfy real-time constraints. When the controller is started, two different methods are executed:

- communicationThread(). An RTAI thread that reads the reference by invoking `getReferiment()` method of the corresponding trajectoryImpl object, and writes information in an attribute of the controller class. This sequence is repeated during each control cycle at every $T_{com}$ $ms$ defined by the user.
- controlThread(). An RTAI thread that reads the reference in the controller class attribute, computes the correct signal (for example torque/voltage signal) and sends it to the robot using the specific board object. This sequence is repeated cyclically at every $T_{ctrl}$ $ms$ defined by the user.

The threads above use a variable represented by the controller attribute object that includes the controller reference; this is a shared variable that must be synchronized using a semaphore paradigm (Penelope uses RTAI semaphore services).

### 3.4 Penelope for Teleoperation

When Penelope is used to manage a teleoperation system the Penelope process must be running. In the case of multiple robots, sensors or general devices, it is possible to use more PC stations, each one running a copy of Penelope. In this case, in any instance of Penelope, only a subset of the system objects is created and the communication is managed by CORBA through the Naming Service and the "creator" object. In other words, any station generates only the objects (for example its robot, board and controller object) relevant to its operation, but it can also use objects of other instances of Penelope, by using the Naming Service. In the GUI, it is necessary to choose which object is created in a specific Penelope instance, and which distributed object is used by the other Penelope instances. To use them, it is necessary to obtain only the reference through object names stored in the Naming Service. Briefly, when Penelope is running, through a GUI it is possible to:

- create new objects on a PC and add new object references in the right Name Service context (for example, create an instance of a robot, of a controller or of a board);
- use objects created to control a robot or for teleoperation experiments (for example, reading joint position or setting torque voltage on a specified motor joint);
- use distributed objects via the Naming Service structure (for example, read joint positions of a robot or read force sensor data acquired by a different PC);
- destroy objects and delete references in the Name Service structure (for example, destroy the current robot instance to use another robot or destroy all objects and delete all Naming Service references to restart a test with Penelope).

In a teleoperation system, Penelope is robot independent since it can control either Master or Slave. The only difference is the configuration procedure

when specific objects are created and the communication channel is established, setting the right configuration parameters in the architecture by using a GUI. It is important to understand that when more instances of Penelope are running, only one Naming Service tree is used. Every Penelope instance uses the same tree with the same structure but in different contexts, as shown earlier in Figure 5.

## 4 Performance Analysis

In [Ark98] Arkin gives a list of desiderata for behavior-based architectures to measure an architecture's utility for a specific problem. This list contains items such as: support for parallelism, hardware targetability, niche targetability, support for modularity, robustness, timeliness in development, run time flexibility, performance effectiveness. These criteria are used in Arkin's book to compare different types of architecture, focusing on mobile robots. In [OC03] a comparative study is carried out of a few successful software architectures for mobile robots, based on features such as portability, ease of use, software characteristics, programming and run-time efficiency. In the evaluation process the authors assess software and programming characteristics of the test system: OS, language support, standard libraries, communication facilities and their performance, hardware abstraction, porting and application building, documentation and programmer efficiency. Then an evaluation of the run-time performance is given, based on application dependent goals, such as localization, mapping and planning. In this approach to system evaluation, it is difficult to measure the coupling between the specific program developed for a task and the software framework that the program is running in.

We can address this problem by proposing the strategy that we use to evaluate our architecture. In order to understand what part of system performance is due to the task-specific algorithm, and what is due to the architecture itself, we measure some of communication times on the critical data paths, i.e. between software modules instead of the more typical "beginning-to-end" time of a complete task. We test the real-time performance degradation of the architecture by making the processor busy with other tasks, and by observing its response to higher priority jobs. In order to develop a repeatable test, we setup a meaningful benchmark that can demonstrate the benefits of an architecture for teleoperation. We will also consider development time as a measure of the environment quality and of the support provided to developers by a structured architecture.

These tests lead to the conclusion that the global evaluation of a software architecture must be user referenced, and should consider performance effectiveness, timeliness in development, support for modularity and hardware targetability. The other criteria indicated in the literature are more appropriate to robotic applications, but not strictly depending on a software architecture. For instance, with respect to the very important criteria of run time flexibility

and robusteness the framework has to take into account reconfigurability and fault recovery, but these problems are mainly matter of good programming practices than of the overall architecture.

### 4.1 Test Results

In order to test Penelope and evaluate the performance of our architecture a teleoperation test is presented. To define a demanding test task, we use the maximum number of classes implemented in Penelope in a time critical operation. We consider a simple teleoperation task with force feedback. We use a NASA-JPL Force Reflecting Hand Controller (FRHC) as the system master. The FRHC, shown in Figure 7, is a six degree of freedom (dof) joystick with force feedback in every joint. It permits to operate with full dexterity in a cubic workspace of 30x30x30 cm developing forces up to 10 N and torques up to 0.5 Nm. The joystick is driven by a custom designed controller implemented mostly on a FPGA board (NI PCI-7831R).

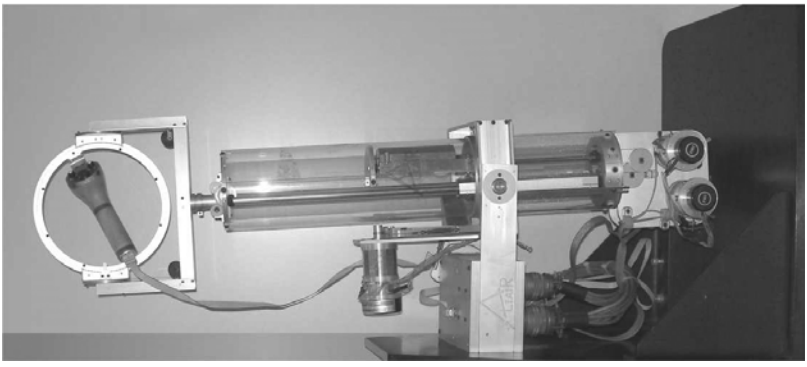

**Fig. 7.** The NASA-JPL Force Reflecting Hand Controller (FRHC).

The idea behind this appraoch is to improve the computational speed of the master while maintaining accuracy and integrating the controller in the architecture. The FPGA is a good instrument to test hardware task implementations at a reasonable cost, reducing the time between the system design and final implementation. Our approach relies on porting the most time-consuming teleoperation algorithms to hardware. In a force feedback teleoperation task, the study of the structure of the joystick highlights that the main computational bottlenecks are the forward kinematic computation and the force transformation from the sensor frame to the joint frame of the joystick. Our FPGA permits to implement in hardware the forward kinematic algorithm. The focus here is on ease of software and hardware integration: the developer has only to maintain the interface expected by the IDL in order to

make the hardware implementation as transparent as possible to the overall software architecture.

The slave robot is a Unimation PUMA 200 arm with six dof, controlled by a Galil board, shown in Figure 8. We use a virtual sensor to simulate different force/torque profiles reflected to the user. Every movement the operator imposes to the joystick is transferred to the robotic arm with the appropriate workspace transformations and scaling. The readings of the robot sensors are fed back to the joystick and, through the activation of the FRHC motors, to the operator. The communication is carried out on¡ a standard 100MB ethernet LAN.


**Fig. 8.** The Unimation PUMA 200.

We run several tests to "tune" and evaluate our architecture. Most of them study the operator's perception, the stability of the entire system and the correctness of the movements of the teleoperated robot. With the final setup, operators have good perception of precision positioning and force reflecting.

Since it is difficult to quantify the architecture performance from these usability tests, we also set up two other experiment groups. The first one investigates the performance of the communication and the control implementation in Penelope. The second group focuses on the advantages of a modular and transparent design with respect to data exchange.

One of the tests objectives is to evaluate whether the use of CORBA adds a significant communication overhead with respect to more traditional communication strategies such as sockets. A CORBA version of `PING` has been implemented on top of our architecture and results are compared with the traditional `ICMP PING` of the Linux operating system. The comparison however, is not completely straightforward. In fact, CORBA supports only TCP and UDP protocols, so a real CORBA-based PING is not feasible. To compare equivalent methods then, we added a similar functionality on top of Penelope using TCP, paying attention to use IP datagram of the same size of ICMP echo-request packets. Instead of waiting for an echo-reply we use the ACK of TCP flow control. The packet size of the acknowledgment is smaller, but this compensates the overhead used by the more reliable protocol. The test is repeated on our laboratory LAN under different load conditions: with a dedicated point to point link between two computer, with normal intranet and Internet traffic and with an overloaded LAN, as shown in Figure 9.

To analyze data we have computed the mean and variance of all data sent from the two PC's; this is well represented by the normal distribution shown in Figure 10 where black distributions refer to normal PING's while red distributions refer to CORBA-based PING's.

We can see that CORBA-based PING is slower than normal PING and this difference is easy to see specially with the PC called Hector. In this case, in fact, the mean of CORBA-based PING is higher than of the ICMP PING of the other PC: with respect to CORBA-based PING, Hector's mean is about $1.7596ms$ and Vincent's mean is about 2.1443, whereas with Linux PING, Hector's mean is about $1.5726ms$, which is very similar to Vincent's mean ($1.5359ms$). This can be explained by considering that the overhead introduced by CORBA has a larger impact on a slower PC than on a newer and faster hardware.

The results of our PING tests represent a lower bound on the communication velocity achievable by our architecture during teleoperation tasks. The ICMP PING provides a measure of fast, but unreliable, communication.

Another important performance measure refers to the maximum frequency of the control loop achievable under RTAI with our hardware. This information alone is not completely significative, because real-time is not synonymous of high frequency, thus the second step is to verify that this performance is guaranteed by the real time operating system even under heavy system load

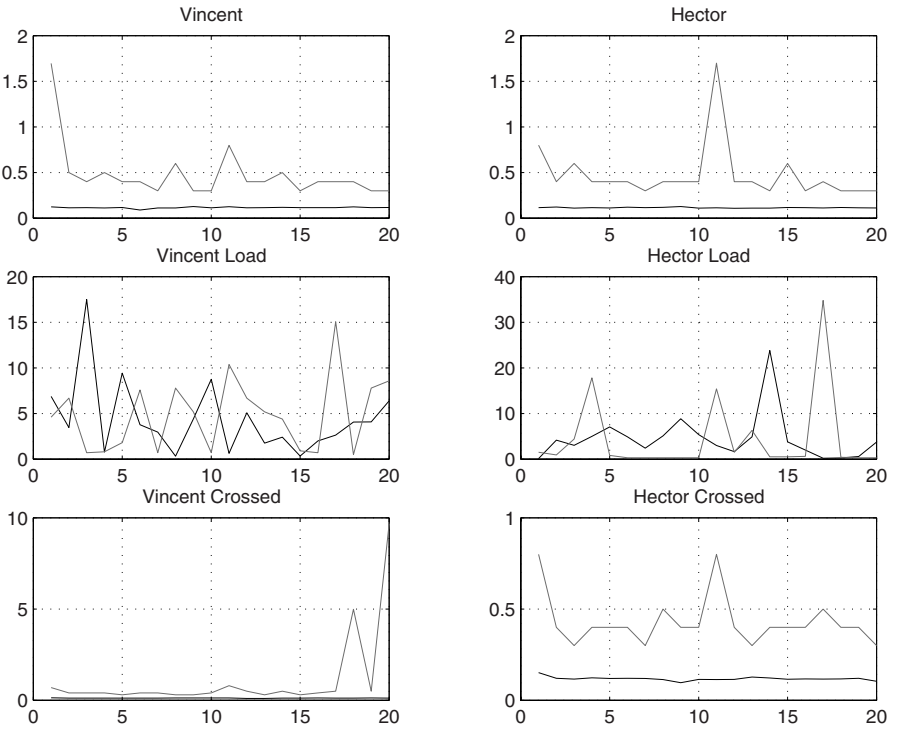


**Fig. 9.** Test communication performance with different PC (Vincent: Pentium 3, 866Mhz, with 256M RAM, Hector: Pentium 4, 2.4Ghz, with 512M RAM). Figures in the first line are with normal laboratory traffic; figures on the second line are under overloaded traffic and on the third line with point to point dedicated connection (Crossed). Black lines represent traditional ICMP PING in Linux while red lines represent CORBA-based PING. All data are expressed in milliseconds.

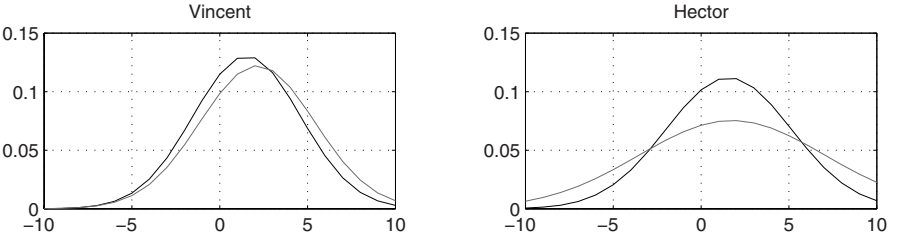


**Fig. 10.** Mean and Variance expressed by normal distributions. All data are in milliseconds.

or typical high latency I/O operations. Once the maximum control loop frequency is known with respect to the hardware limits of our host, a hard real time task with this time constraint is started. The computation must work without loss of performance while the system is loaded with different applications and data transfers, as shown in Figure 11.

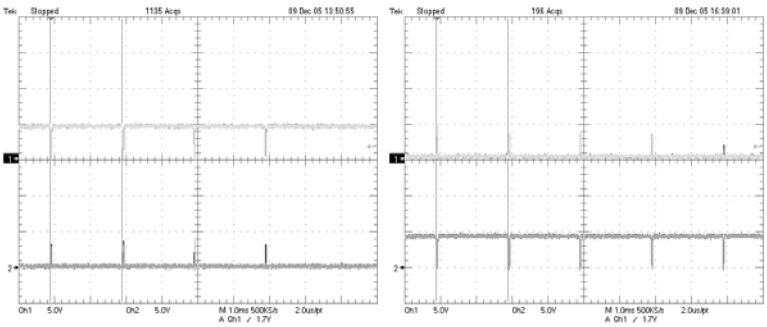

**Fig. 11.** RTAI performances with Hector PC with normal load (left figure) and with heavy system load (right figure) with a period of $2ms$ measured by Digital Oscilloscope. Every peak is a pulse train (25 pulses). No performance loss is visible.

The second group of tests is designed to demonstrate the modularity and reusability of Penelope. In these tests we want to exploit the capability of CORBA of handling distributed components. Different configurations have been distributed over our system paying attention to how many reconfigurations or modifications, with subsequent recompilations, Penelope required. We first tested the standard teleoperation task in a point to point connection, one PC controlling the joystick, collecting movements and driving the motors for the force feedback, and the other PC controlling the robot arm, simulating the force/torque sensor and running the CORBA naming service. All the matrix transformations needed by the FRHC and the robot are carried out by the respective hosts. Then, a different setup is used. Four computers are involved in the task: one for every device (robot, joystick and sensor) and one for the CORBA naming service. The performance of the two systems are compared.

One important fact arising from these tests is that no recompilation is needed. The selection of the service needed in the task is made at run time. CORBA takes care of identifying the host that can provide the elaboration requested in a very transparent way. If we decide to change the control type of one of our devices, for example transferring the forward kinematic computation of the FRHC to our FPGA to speed up the computation, only the selected classes on the selected host needs to be changed. Intercommunication interfaces are already defined and coded into the IDL.

Another test that involves both performance and modular capabilities of Penelope is to read sensor data from two different computers. Even in this case no change is needed to the code. The GUI's of the two hosts permit to choose the same service and no delay in the communication is introduced by the parallel requests to the sensors. Each device is read once and then transmissions are made in parallel over the net but not high load is detected on the net, according to our monitoring devices.

## 5 Conclusion and Future Work

In this Section, we have described Penelope, the architecture developed at the Robotics Laboratory ALTAIR of the University of Verona (Italy) and we presented the results of preliminary tests on a real teleoperation system. The tests are very encouraging and suggest that more devices and software algorithms can be added effortlessly to the system. Our goal is to demonstrate the feasibility of the solution proposed for a global system structure in a research laboratory where fast prototyping is a very important development aspect, together with performance improvement, especially with respect to the system real-time behavior. We want to have a common software basis to support the migration of algorithms across the laboratory and to our technology transfer partners. The use of a single common architecture would have a tremendous impact in this specific environment. We do not think that a unique framework can be possible across laboratories because from our experience a performing architecture is strictly tied to specific hardware and software. However, we think that it is possible to exchange high level modules when the architecture is truly modular.

RTAI and CORBA are difficult instruments to learn and to use. A lot of time has to be spent for an in-depth understanding of their mechanisms and functionalities. However, the design of Penelope and the current implementation relieves the programmer from a heavy involvement in communication and timing. For example, if a new type of controller is coded in our architecture it will perfectly fit in the already made IDL definitions, and the structure of communication/control synchronization may be reused.

Hard real-time thread must be free of unreliable instructions such as device driver access and network direct utilization. That's way the communication and the control loop are decoupled. Specific software is available in order to use simple, but widely used, communication channels with deterministic timing capabilities. USB4RT [Kis] and RTNet [Mar] are projects that will permit the integration among USB, ethernet and RTAI. Serial and parallel port support are already part of RTAI. Specific devices, such as axis boards, need kernel modules customization in order to achieve this goal.

One of the most important open issues regards the synchronization of data exchanged through CORBA. Due to the distributed nature of Penelope the possibility that each sensor is driven by a different PC has to be taken into

consideration. If the teleoperation task needs the utilization of both cameras and force sensors, a synchronization mechanism must be developed to ensure that streaming video and force feedback reach the operator in a synchronized way, or a loss of realism will occur. The solution to this problem is not simple because of network uncertainty and data acquisition speed. We are working on a heartbeat strategy to maintain a common clock on the distributed environment. To achieve this goal we want to define a high priority channel using RT-CORBA, but we must still develop this technology. Some new real-time features like RT-Net may be useful even if these "drivers" rely on specific chipsets of ethernet board and are mostly for point to point connection. A different approach consists in using an NTP (Network Time Protocol) such as the one described in [Pro].

A more precise and general evaluation testbed for this type of architectures is needed, to make comparisons between different strategies easier and more useful. We are addressing this problem while our architecture becomes more usable and stable, letting us focus on performance and task related problem.

Finally, we are implementing a web interface for Penelope. The objective is to use it from Internet in various types of robot control experiments. User will be able to connect with Penelope, choose the controller type and its parameters, the trajectory type and its parameters, and run the remote experiment sending command through Internet. We have only added an IDL class (webInterface) and implemented the correspondent C++ class (webInterfaceImpl). In this manner, a client must only know this class interface and not the entire Penelope structure. In the same way, additional external tools or systems will be able to communicate with Penelope either using the standard Penelope interface (with all classes, methods and attributes) or using a built-in interface to hide complex structure to the client (for example with Matlab or Simulink toolboxes).

**Acknowledgements**

The authors would like to thank Davide Moschini for his participation in the design and in the early development of Penelope.

## References

[Ark98] Ronald C. Arkin, *Behavior-based robotics*, The MIT Press, 1998.

[AS92] R.J. Anderson and W Spong, *Asymptotic stability for force reflecting teleoperators with time delay*, Journal of Robotics Research **11** (1992), no. 2, 135–148.

[Con] Scilab Consortium, *Scilab - a free scientific software package.*

[Des] Mathieu Desnoyer, *Ltt next generation (lttng).*

[DIA] DIAPM, *Real time application interface (rtai).*

[DZK92] H. Das, H. Zak, W. S. Kim, A. K. Bejczy, and P. S. Schenker, *Operator performance with alternative manual control modes in teleoperation*, Presence **1** (1992), no. 2, 201–218.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable software architecture*, Addison-Wesley, 1995.

[Gro] Object Management Group, *Uml: Unified modeling language.*

[HV99] M. Henning and S. Vinosky (eds.), *Advanced corba programming in c++*, Addison Wesley, Massachusetts, 1999.

[Kis] Jan Kiszka, *Usb real time support (usb4rt).*

[Law93] D.A. Lawrence, *Stability and transparency in bilateral teleoperation*, IEEE Transaction on Robotics and Automation **9** (1993), no. 5, 624–637.

[Mar] Ulrich Marx, *Rtnet hard real-time networking for real-time linux.*

[Mat] MathWorks, *Matlab - the language of technical computing.*

[OC03] Anders Orebaċk and Herik I. Christensen, *Evaluation of architectures for mobile robotics*, Autonomous Robots **14** (2003), 33–49.

[OMG] Object Management Group OMG, *Common object request broker architecture*, Tech. report, http://www.corba.org.

[Ope] Opersys, *Linux trace toolkit (ltt).*

[Pro] Network Time Protocol.

[Ric99] Charles Richter, *Designing flexible object-oriented systems with uml*, Sams, 1999.

[She92a] T. B. Sheridan, *Telerobotics, automation, and human supervisory control.*, Mit Press, Cambridge, MA, 1992.

[She92b] T. B. Sheridan (ed.), *Telerobotics, automation and human supervisory control*, MIT Press, Cambridge, MA, 1992.

[YY94] Y. Yokokohji and T. Yoshikawa, *Bilateral control of master-slave manipulators for ideal kinestetic coupling–formulation and experiment*, IEEE Transactions on Robotics and Automation **10** (1994), no. 5, 605–620.