INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

GRADUATE PROGRAM IN
MECHATRONICS AND INFORMATION TECHNOLOGIES

A WRAPPER COMPONENT-BASED METHODOLOGY FOR INTEGRATING DISTRIBUTED ROBOTIC SYSTEMS

THESIS

PRESENTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE ACADEMIC DEGREE OF

DOCTOR OF PHILOSOPHY IN
ARTIFICIAL INTELLIGENCE

BY

FEDERICO GUEDEA ELIZALDE

MONTERREY, N.L., MÉXICO , MAY 2008

A Wrapper Component-Based Methodology for Integrating Distributed Robotic Systems

A Dissertation Presented by

Federico Guedea Elizalde

Submitted in partial fulfillment of
the requirements for the degree of

Doctor of Phylosophy
in the field of
Artificial Intelligence

Thesis Committee:

Fakhreddine Karray, University of Waterloo
Rogelio Soto Rodríguez, ITESM Campus Monterrey
Jose Luis Gordillo Moscoso, ITESM Campus Monterrey
Ricardo Ambrosio Ramírez Mendoza, ITESM Campus Monterrey
Santiago E. Conant Pablos, ITESM Campus Monterrey

Center for Intelligent Systems

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

May 2008

Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

Division of Mechatronics and Information Technologies

Graduate Program in Mechatronics and Information Technologies

The committee members hereby recommend the dissertation presented by **Federico Guedea Elizalde** to be accepted as a partial fulfillment of the requirements to be admitted to the **Degree of Doctor of Philosophy in Artificial Intelligence**.

Committee members:

Dr. Rogelio Soto Rodríguez
Advisor

Dr. Fakhreddine Karray
Co-Advisor

Dr. José Luis Gordillo Moscoso
Member

Dr. Ricardo A. Ramírez Mendoza
Member

Dr. Santiago E. Conant Pablos
Member

Dr. Joaquín Acevedo Mascarúa
Director of Research and Graduate Studies
School of Engineering

May 2008

Dedication

This thesis is dedicated to my wife Isabel Cristina, my son Omar Moisés, my parents Melchor and María Jesús, my brothers and sisters, and my in-laws for all the unconditional confidence, support, patience, encouragement and motivation they provided me for pushing through this work.

Declaration

I hereby declare that I composed this dissertation entirely myself and that it describes my own research.

Federico Guedea Elizalde
Monterrey, N.L., México
May 2008

Acknowledgements

I would like to thank Dr. Karray for several reasons. Firstly, he allowed me to continue my doctoral studies as a visiting scholar in Systems Design Engineering Department and Pattern Analysis Machine Intelligence Laboratory at the University of Waterloo (UW) for more than 2 years. Secondly, as my co-advisor, he provided access to many vital resources such as current related journals and publications, software, and the arm manipulators and mobile robot which played an important role in part of my research.

I would like to thank Rogelio Soto for allow me to work in this field. He granted me the freedom to explore various frameworks when trying to define my research project and he helped me to secure a position as a visiting scholar at the University of Waterloo.

I would also like to thank my committee members Dr. Jose Luis Gordillo, Dr. Ricardo A. Ramírez and Dr. Santiago Conant Pablos for taking the time to review my thesis. Their support was very important. Dr. Gordillo and Dr. Conant provided many recommendations to improve my final writing.

The doctoral program coordinators Rogelio Soto and Hugo Terashima also helped me on numerous occasions.

I would like to thank Insop Song and Haikal Alhichiri whom I feel fortunate to have met. During my years at UW, Insop was an incredible colleague who supported me in my work. His support and friendship was invaluable. During the following years his motivation and ideas were essential to my research and I have the fortune to work in several projects with him.

During my stay at Canada I met many persons, but certainly my family and I are very happy to know John & Sydney Henderson. They were and they are our family in Canada. We were blessed with their unconditional love, help and guidance. Through them we met other persons who also gave us all their love, such as Jack & Betty Strauss and Ron & Dorothy Worth.

I would like to thank to my family at Monclova: My parents *Melchor* and *María Jesus*, my brothers and sisters *Isabel*, *Melchor Jr.*, *Irene*, *Sergio* and *Jorge*, whose love and support I could always count on. Also, thanks to *Lichis* for her unconditional caring and her parents *Don Cande* and *Doña Hermelinda*.

Finally, I would to thank to the most important people in my life, my wife and my son. Thanks *Cristy* and *Omar Moisés*, for giving me new dreams to pursue. I am always so proud of all my family. They were my main motivation and inspiration each and every day that allowed me to continue and finish this dissertation.

Abstract

Building an intelligent robot system has been an extensive research area. There are many advances in components needed to construct a robotic system, such as vision systems, sensory systems, and planning systems among others. Integration of these components represents a big challenge for robot designers, because each component comes from different vendors and they run with different interfaces or under different operating systems. This will be even more difficult if the overall system development has to deal with environmental uncertainties or changing conditions. In these cases, new tools and equipments are necessary to adapt the initial configuration to the new changing requirements. Each added component increases the complexity of integration due to the interconnection required with the previous components.

This thesis research presents a novel approach to solve the integration problem using the concepts of distributed framework and distributed computing systems. We named this methodology DWC (Divide-Wrap-Connect). This methodology is based on the mechanisms used by standard middleware software to provide transparency and portability among different operating systems and languages. The idea is to define software modules named *Wrapper Components*, which are object-oriented modules that create an abstract interface for a specific class of hardware or software components. These modules are the components of a bigger system.

Basic steps in this methodology are: a) Divide, b) Wrap and c) Connect. The creation of Wrapper Components is the core activity of the second step (b) but its design is affected by the first and third step of this methodology. We provide some basic definitions in order to clarify the scope of different design alternatives. Furthermore, we present how using standard mechanisms from distributed computing such as Event services and Naming services, the third step (c) is improved.

We tested our approach by solving an experimental classical AI problem, block-world problem. The intelligent functions are object recognition, environment recognition, planning, tracking capabilities, tasks control and robot arm control. These functions were developed into several components and a coordinator module. This coordinator modules interacts with the user and the other components in order to solve the block-world problem. The construction of the system was done in an incremental way showing the benefits of this methodology.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation presents a methodology to integrate a variety of applications, tools and systems from different disciplines such as Distributed Computing, Computer Vision, Artificial Intelligence and Robotics.

Nowadays, there is a plethora of robotic systems which are configured with a privately owned closed architecture. Integrating these systems with other components such as vision, sensors, planning systems, factory database information systems is a challenge that requires specific ad-hoc design. The interfaces between components are such that any change in the configuration or specification of the overall system requires redesign and implementation.

Researchers have proposed different alternatives to integrate robotic systems. These are designed to work for specific tasks and environments but have constraints and limitations that make it difficult to integrate new robotic components (Chapter 2.0).

In this chapter we outline the main motivations of the research, the specific problem to be addressed, the methodology we are proposing, the scope of this thesis, and an outline of the dissertation.

1.1 Motivation

This research is addressed to create a Distributed Robotic System (DRS) that has better performance with less effort of integration. Then, why is this necessary? Certainly, there are many types of robotic systems nowadays, and each one is designed with a specific task in mind. However, there are some areas that can be benefited by using new tools. The following is a list of these areas:

- There are many real problems that can not be solved by only one entity, but they can only be partially solved by many entities that interact under a common global goal or goals. In these cases it is very difficult to find an entity that knows all the details that concern the main problem. In other words, each entity has a limited knowledge about all problem details that arise in the time. Some examples of these systems would be a team work or any other team with multiple players, i.e. a production line, the construction of a building, a manufacturing robotic cell, and others. In these examples, a task division can be visualized based on some specialized skills and/or hierarchical order. The later is necessary in order to coordinate and to control the multiple tasks realized by each entity. Figure 1.1 shows a complex process with multiple single processes interacting. One of the main methods of solving complex problems is to divide them in some way in order to create modular components. These modules can be coordinated through manager or supervisor components that communicate with each other. Figure 1.2 shows this behavior. Even though, the design for these main components is made to achieve a predefined behavior or to meet a specific goal. Any system deviation from this behavior is seen as a perturbation and human operator intervention is required. In other cases, the goal can be changed and a new program or design is required, so the human intervention takes precedence again, but now from the system designers.

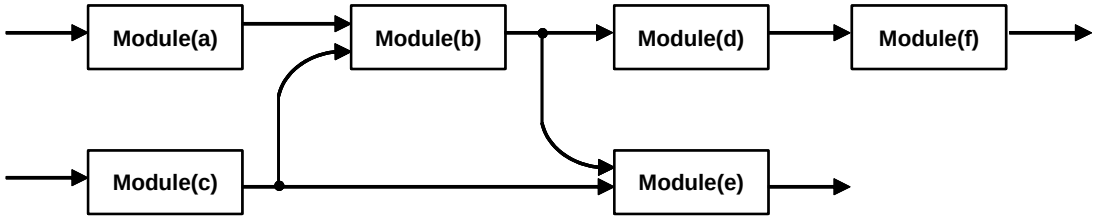


Figure 1.1: Complex process with multiple single process interacting.

- Developing large and distributed robotic applications is a very challenging job. If this work is carried out in university laboratories then the challenge is increased. Laboratories tend to acquire robotic equipment for specific projects and sometimes with limited budget. Also, the lab is used by different team projects. All of the above situations create a mixture of equipment from different vendors that are not compatible, or even worse, totally incompatible. Then, how can this robotic equipment being integrated with the other robots in the lab?

- There are few facilities with suitable environment for the development, implementation and evaluation of components, protocols, and operational modes required on tele-robotic systems [Skubic *et al.*, 1995]. The research on robotic system integration issues has been slow to assess the impact of new component technologies.

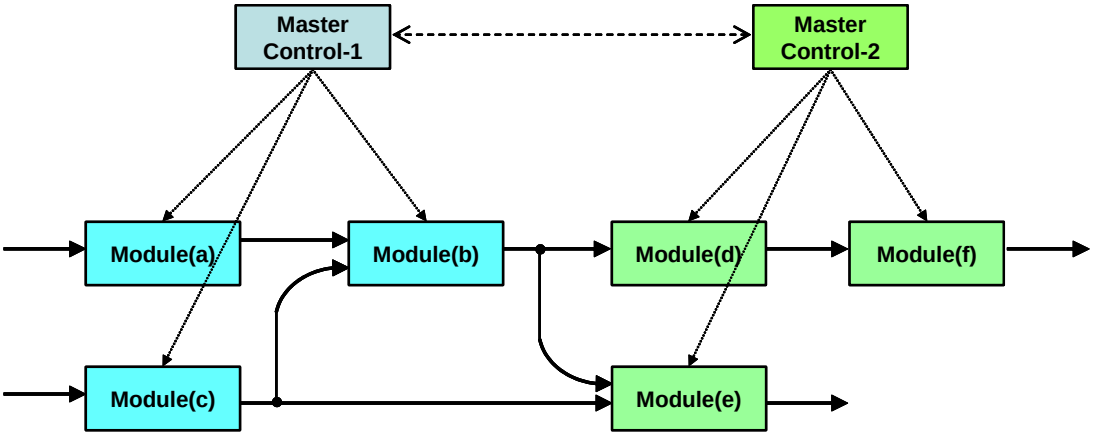


Figure 1.2: Modular manager components (Master controls) applied for complex process.

Furthermore, the technology for creating such environments is relatively immature. Most of the current capabilities are based in point-to-point remote operations and have been proven to be useful research environments. However, ground-based applications (e.g. hazardous operations), space applications (e.g. ground-to-air control), and the ground-based development process (including evaluations) require system configurations with greater degrees of freedom, i.e. connectivity of multiple sites with different functions.

- In countries with less infrastructure and resources, it is desirable to link facilities and resources from different sources, such as government, industry, and the university, so that a collaborative research and development environment can be realized with associated sharing of resources [García-Zubia *et al.*, 2005; Esche *et al.*, 2003; Trevelyan, 2004].

  This will enhance the educational setting available to graduate students, or will help the industrial companies with specific experiments, where they can monitor closely but without need to stay on site. DRS research is a good basis to link research at different locations. It enables students to access equipment that would not be possible to access from their local universities [Deniz *et al.*, 2003; Casini *et al.*, 2003; Wong *et al.*, 1999; Benmohamed *et al.*, 2004].

- Industrial applications using robotic systems have demonstrated their utility in many repetitive tasks, such as car manufacturing, production of pharmaceutical and packaging systems. In these cases the robotic environment must be very structured to avoid failures. However, there are other applications where the tasks are ill-defined or the environment is unknown to a certain degree, such as space exploration or hazardous operations (e.g. bomb deactivation, management of radioactive materials, etc.). Because of these factors, the robotic application

must have a greater deal of surviving under these circumstances. Integration of more sensory systems is necessary, in addition to better algorithms that increase the dexterity of the robot. This represents a challenge, and a time consuming task for the robot designer, it is mainly because of the closed controller architecture of the robotic components. Certainly, it is desirable to have a plug-and-play component that can be integrated in several minutes rather than days or months.

All of the above examples demand new tools or approaches from areas other than robotic control and automatic control systems. These must be included in the design of distributed robotic control systems, in order to incorporate intelligent components that improve the overall system performance. The main questions motivating this research are:

- *How to integrate different types of robotic components?*
- *What is the structure that these components must have to work as a team?*
- *What are the tools that these systems must have to adapt for uncertain environments?*

1.2 Problem Statement

Distributed Robotic Systems (DRS) is a very wide and multi-disciplinary research area. The applications in this area vary from (i) operation of autonomous vehicles in different types of environment, (ii) the organization of tasks for modular or reconfigurable robots, (iii) distributed intercommunication units, just to mention some examples. Generally, it is expected that DRS is capable of the tasks that are impossible for single robots (such as carrying objects that are large compared to the scale of one robotic unit), and it is more reliable and as well be self-repairable due to their modularity of construction, and in addition to reduced cost. Service robots interact closely with humans in a wide range of situations, applying both skills and knowledge to cooperative tasks. Thus, the robot control systems must be equipped with the resources for intelligent action. These resources are presented in the modern robotics literature as separate ideas, solutions, and mechanisms. What is needed is the architecture for coherently combining them into an integrated system [Beni and Wang, 1991].

Because of the large number of related fields with common concepts, it is easy to see that many problems arise when designing a DRS. More specifically the following problems are among the most often considered in relation to the design:

- Communication among intelligent robotic components.

- Physical connection of autonomous modules.
- Reconfiguration of the system to deal with new environments.
- Methods of distributing intelligence and control among the components.
- Actual construction of the robotic components (hardware implementation).
- Hierarchical control.

From this list, the most important item is the construction of the robotic units. This step determines the scope of the other related problems. The second is the emphasis on communication, i.e. creating effective means for the units to form a cooperating system. The third is the emphasis on algorithms, i.e. realizing ways of accomplishing the tasks. In this thesis, the communication problem is addressed as part of the integration problem. The solution to the communication problem is significantly dependent on the physical construction since it is usually difficult to port communication systems designed for a specific hardware into another. Besides the physical dependency, the solution of this problem must provide a platform for the next stage, i.e. the realization of distributed algorithms or distributed software components.

1.3 Thesis Statement

In order to provide an effective solution for the *integration* and communication problems when designing a DRS, an Integration Methodology is proposed. This methodology is based on concepts from the area of Distributed Computing. In this Thesis, we propose the use of *Commercial-off-the-Shelf* (COTS) middleware components to construct a communication framework for DRS. Specifically it is the use of mechanisms created by the area of distributed computing. These mechanisms offer the advantage of using the best software methodologies for a set of well-known distributed computing problems and provides solutions in a modular way. the *Common Object Request Broker Architecture* (CORBA) specification [OMG, 2000] is one example. This specification is Object-Oriented and is very suitable for robotic applications. Many researchers from different research areas have been improving the CORBA specification in order to match specific problems found in the daily communication process.
By using standard specifications the application designer does not have to deal with all intrinsical details of communication problems, but it has to create software components that can be reusable, easy to modify, configurable, easy to maintenance and easy to *integrate*. In this sense, although it is possible to use a solid middleware specification, it doesn't solve the integration between components just by applying the specification.

Usually, software packages are libraries provided by the software's vendor or programs created by other persons in the company, or they come with the equipment (i.e robot software). Each package provides a specific interface to interact with it and in many cases all methods or functions provided are not used totally. To face all these aspects is necessary to *encapsulates* the main issues of each package and put them in modules that can be easily accessed by all entities into a system. Our proposed methodology is focused in these issues.

1.4 Methodology proposed

This dissertation proposes a methodology to *integrate* robotic components provided by different vendors, and which software modules could run in different platforms and different operating systems.

The methodology consists of three phases or steps: *Divide, Wrap and Connect*. See Figure 1.3. We named it *DWC* and it is based on *Wrapper Components* (WC). A Wrapper Component is an object-oriented module. It encapsulates the main issues and functions of each robotic component of the DRS. The main attribute of a Wrapper Component is its abstraction level definition. At high level abstraction the Wrapper Component offers a better structure for easy integration and reuse. At low level abstraction the Wrapper Component is more difficult to exchange or modify. To implement these Wrapper Components we selected CORBA specification as the software bus to communicate them (to connect them). Although, similar middleware specifications or products can be used, such as ICE from zeroC [ZeroC, 2002],[Henning, 2004]. This speeds up the development time due to they are using the same communication bus but there are also other aspects that we must take into account for a better performance. The loosely coupled and asynchronous operation of a component simplifies the system model at a high level. Over-specification at system's higher level can lead to a non robust operation; thus a collection of asynchronously executing components is more stable.

We propose the use of Wrapper Components to construct and to integrate distributed robotic systems. These components are built based on CORBA specification and CORBA services. Under this scheme, the wrapper components are defined through the Interface Definition Language (IDL) provided by CORBA specification, see Figure 1.4. The main idea behind this approach is to go farther than just creating simple IDL interfaces for each robotic component. For example, instead of creating a specific interface for each function of a determined component, it is better to define a set of abstract functions for similar robotic components. Although the idea looks simple the implications are not simple, but the final product is easier to be connected into the whole sys-

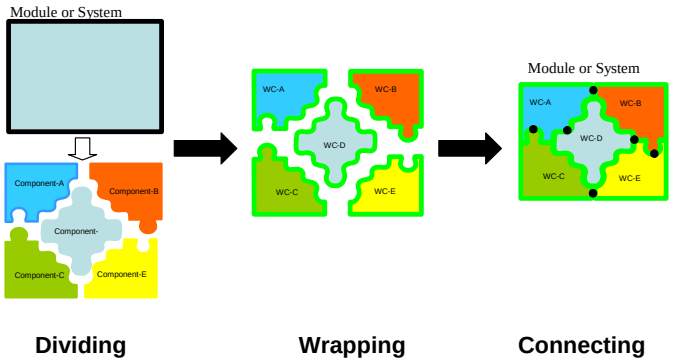


Figure 1.3: Basic steps on DWC methodology

tem. We proved these properties during the implementation of the CONCORD project [Song *et al.*, 2004].

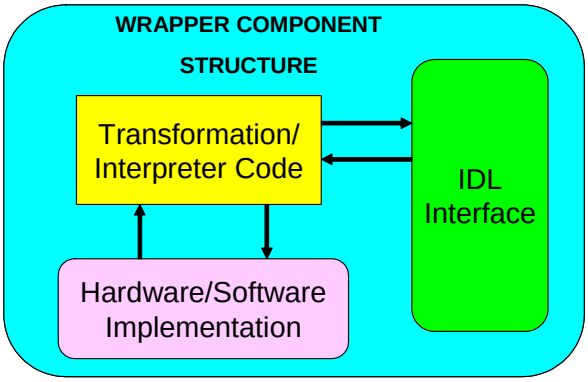


Figure 1.4: Proposed structure for a Wrapper Component

1.5 Scope of the Thesis

The research performed in this work concentrates on the issues involved during the creation of Wrapper Components. We define the internal structure of a Wrapper Component and we explain how this component could be easily replaced or updated without modifying other modules of the distributed robotic system. The main contributions are on the structure of the Wrapper Components and the mechanisms used to connect them according to the tasks to realize.

We concentrate our research in the creation of Wrapper Components for three main components involved in the creation of Intelligent Distributed Robotic systems: Robot Server, Vision Server and Planning Server.

We tested these components in an incremental approach by integrate them in several

ways to perform different tasks. First we created a Remote Robot Manipulation system in which the intelligence is on the human-side and CORBA middleware is used to provide the communication mechanisms for robot commands and visual feedback. In this case the interface for a Robot Server is defined. Second, we concentrated on the creation of a Vision System. This system is able to be operated remotely and it can learn several objects selected by a human operator. Once the objects are learnt the system can find them and track them. In this stage a Vision System Server is defined. Third, the previous two systems are integrated together with a planning system and a coordinator component to provide an autonomous robot operation. This autonomous behavior is tested using the classical block-world problem on Artificial Intelligence.

At last, the robot server was used in other two projects using the same wrapper interface. These projects are related to robot collaboration in a master-slave configuration [Li *et al.*, 2005], and remote robot operation through speech recognition following a specific command language for the robot server [El-Khalil *et al.*, 2004].

1.6 Dissertation outline

The body of this dissertation consists of eight chapters and four appendices. Chapter 2 presents a chronological literature review of similar distributed robotic systems designs and implementations, we highlighted the evolving development of these approaches, and how the *Integration* issue is tackled using middleware software. Chapter 3 presents a definition of Distributed Robotic System and its main advantages and challenges. Also a brief description of CORBA specification is presented. Chapter 4 presents the methodology proposed to construct "building blocks" by means of the definition of wrapper components and by using CORBA services. Chapter 5 presents the application of the methodology to define the main wrapper components to integrate a distributed robotic system. Here the methodology exposed in Chapter 4 is applied. Chapter 6 presents the integration experiments and measures of the components defined in Chapter 5 using an incremental approach. This approach exposes the advantage of reusable code and flexibility on the design. This is our main contribution and we explain the key issues needed to be successful when different components must work together to achieve more complex tasks. The dissertation concludes with Chapter 7, which reviews the contributions of our work and discusses interesting directions for future work.

Four appendices provide additional details on features of the implementation and testing domains. Appendix A explains in more details the CORBA specification and some of the services used in this work. Appendix B details how the robot servers are built and

exposes the advantages of an abstract design. Appendix C explains the abstract level of functionality of the vision server for robot applications and how the communication is established to provide almost a real-time response. Although computer vision is a very large research area, there are some constraints in the robotic systems that reduce the usefulness of vision algorithms, so the challenge is to find a way to improve the vision algorithm for this kind of applications. Appendix D provides an overview of the planning system and details how a legacy software planner (Graphplan algorithm) can be wrapped to be used into a distributed environment. This is a key tool to provide certain level of intelligence to the robotic system.

.

Chapter 2

Literature Review

In the literature, there are several approaches to create solutions for distributed intelligent robotic systems. The followings are some of the main works in this field. They are shown in chronological way in order to observe their evolving development. Curiously, much of the work is done for mobile robots and a few ones are developed for arm manipulators.

2.1 TCA

TCA [Simmons *et al.*, 1990]: The Task Control Architecture (TCA), created at Carnegie-Mellon University (CMU) by the beginning of 1990's, simplifies building task-level control systems for mobile robots. Its primary application was focused in the Ambler six-legged walker (funded by NASA). By "task-level", we mean the integration and coordination of perception, planning and real-time control to achieve a given set of goals (tasks). TCA provides a general control framework (shown in Figure 2.1), and it is intended to be used to control a wide variety of mobile robots. TCA provides a high-level, machine independent method for passing messages between distributed machines. Although TCA has no built-in control functions for particular robots (such as path planning algorithms), it provides control functions, such as task decomposition, monitoring, and resource management, that are common to many mobile robot applications [Simmons, 2000].

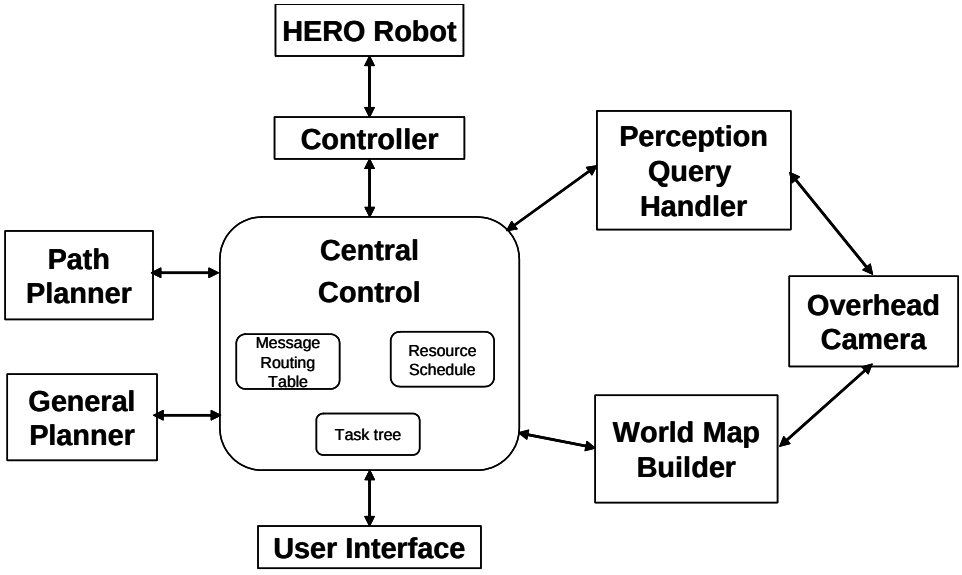


Figure 2.1: Task Control Architecture (TCA) created at Carniege Mellon.

2.2 GLAIR

GLAIR [Hexmoor *et al.*, 1993a; Hexmoor *et al.*, 1993b]: A Grounded Layered Architecture with Integrated Reasoning (GLAIR) for Autonomous Agents was developed by Hexmoor *et.al.*, in the University of Buffalo starting the 1990's. This architecture is based on the concept of embodiment; the process of acquiring concepts that carry meaning in terms of the agent's own physiology. In this sense, the agent's sensors and actuators information is matched against a specific representation based on some extracted features. GLAIR is a general multi-level architecture for autonomous cognitive agents with integrated sensory and motor capabilities. These levels are: Knowledge level, Perceptuo-Motor level and Sensori-Actuator level. The first level corresponds to the conscious part of the robot behavior meanwhile the other two levels work in a unconscious manner. Figure 2.2 shows this multi-layer model. There are several features in the different levels:

- The levels in this architecture are semi-autonomous and work in parallel.
- Conscious reasoning guides the unconscious behaviors and unconscious levels, which are engaged in perception and motor processing.
- Unconscious levels can alarm when an important event occurs and they can take control if necessary.

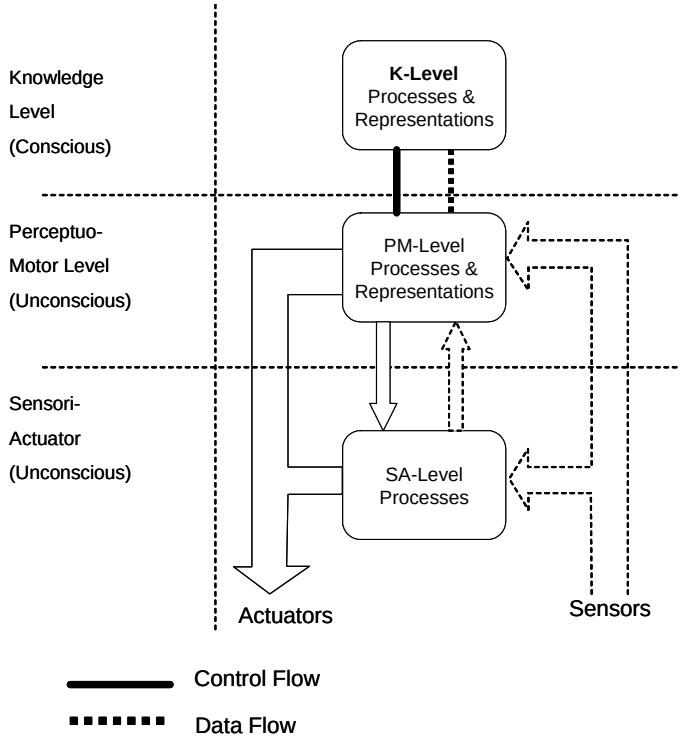


Figure 2.2: Schematic representation of the agent architecture under GLAIR concept.

2.3 CIRCA

CIRCA [Musliner *et al.*, 1993]: The Cooperative Intelligent Real-Time Control Architecture (CIRCA) was created by Musliner *et.al.*, with one objective in mind: combine the reasoning power of unrestricted Artificial Intelligence (AI) methods with the ability to make hard performance guarantees. In this sense, the architecture is divided in two main subsystems: A Real-Time subsystem (RTS) and an Artificial Intelligent subsystem (AIS), as it is shown in figure 2.3. A distinctive feature of CIRCA is that it is based on memoryless and unclocked reactive execution engine, but nevertheless manages to meet hard real-time constraints. The AIS develops executable reactions plans that will assure system safety and attempt to achieve systems goals when interpreted by the RTS. Both AIS and RTS work in parallel or concurrently. This implies a close communication scheme between these two subsystems, but the architecture doesn't mention about the requirements for this connection.

2.4 TelRIP

TelRIP [Skubic *et al.*, 1995]: The Universities Space Automation and Robotics Consortium (US- ARC), formed by several Universities in Texas and the NASA space agency,

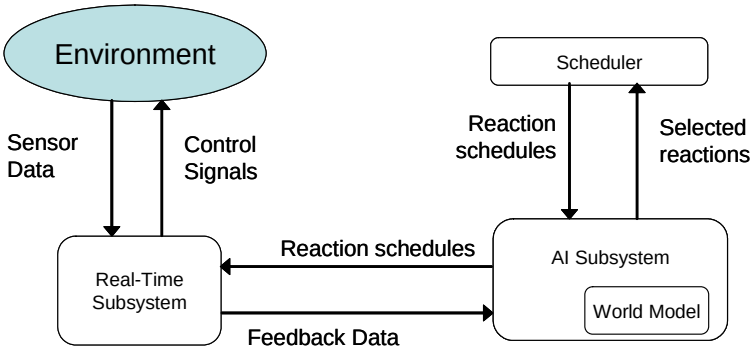


Figure 2.3: CIRCA: The Cooperative Intelligent Real-Time Control Architecture.

has been working in the design and implementation of a Telerobotic Construction Set (TCS) to enable the building of modular telerobotics networks. These modules exchange data using the Telerobotic Interconnection Protocol (TelRIP), which was developed by Rice University. TelRIP is a mechanism that uses the producer/consumer approach to deliver data objects. The main contribution of this mechanism is its capability to measure and monitor the communication performance. This includes message tracing, timestamping, and data logging.

2.5 ISAC

ISAC [Bishay *et al.*, 1995]: In this work, intelligent behavior emerges from the interaction of atomic agents in the Intelligent Machine Architecture (IMA). Each atomic agent acts locally, based on its internal state, and provides a set of services to other agents through various relationships. The communication among the different agents is classified in a) one-way data-flow (or observer) communication for Sensor atomic agents b) master-slave communication for Sequencer agents and c) command-in and position-out communication for Actuator agents.

2.6 AuRA

AuRA [Arkin and Balch, 1997]: The Autonomous Robot Architecture (AuRA) was developed in the mid-1980's as a hybrid approach to robotic navigation. Arkin *et.al.* proposed two main and distinct components, which are a) a deliberative hierarchical planner and b) a reactive planner. The system exposes a collection of behaviors specified and instantiated by the hierarchical planner in the high level of control. These deliberative behaviors are mixed with a basic reactive behavior to control the move-

ment of the robot.

2.7 ARCO

ARCO: Sanz *et.al.*, used CORBA as the middleware for their multi-mobile robot research work in the project called ARCO, Architecture for COoperation of mobile platforms [Sanz *et al.*, 1999]. The authors remark that the issue of Integration is one of the biggest problems to tackle in the development of large and complex systems using artificial components. This problem is addressed using modular, generic, flexible and compatible components. Sanz et.al.,proposed a methodological core called Integrated Control Architecture, ICa [Sanz *et al.*, 2001] based on CORBA characteristics.

2.8 CAMPOUT

CAMPOUT [Pirjanian *et al.*, 2000]: CAMPOUT stands for Control Architecture for Multi-robot Planetary OUTposts. This is an architecture that consists of a set of key mechanism and architectural components to facilitate development of multi-robot systems for cooperative and coordinated activities. These are a) Modular Task Decomposition, b) Behavior Coordination Mechanisms c) Group Coordination, and d) Communication Infrastructure. CAMPOUT has been used for the development of reconfigurable robotic systems [Pirjanian *et al.*, 2002], mainly for tasks in space exploration, such as Mars exploration. The communication facilities are provided using UNIX-style sockets. They consist of the following core functions: Synchronization, Data exchange and Behavior exchange.

2.9 MIRO

MIRO: Utz [Utz *et al.*, 2002], developed the Middleware for Mobile Robots (MIRO) using CORBA. MIRO is structured into three architectural layers: a) The MIRO Device Layer which provides object-oriented interfaces for all robot's sensors and actuators, b)MIRO Service Layer which provides generic services for sensors and actuators through CORBA interfaces and event-based communication, c) MIRO Class Framework which provides a number of functional modules for mobile robot control, such as mapping, self localization, behavior generation, path planning, logging, and visualization facilities. MIRO uses TAO [Schmidt *et al.*, 1997], the ACE (Adaptive Communication Environment) for ORB package implementation of CORBA.

2.10 OROCOS

OROCOS [Bruyninckx, 2001; Bruyninckx, 2002]: Bruyninckx *et.al.*, used CORBA for the Open Robot Control Software (OROCOS) project. OROCOS is an European project, started on September 1st, 2001. Three laboratories are participating: the Katholieke Universiteit Leuven (KULeuven, Belgium, project contractor), the Laboratory for Analysis and Architecture of Systems (CNRS/LAAS, France) and Kungl Tekniska Hgskolan (KTH, Sweden). The project aims at producing an open source software framework for robots, by providing a functional basis for general robot control. The software is intended to be platform independent, but it is also thought of as application independent. OROCOS is based on the definition of generic components, so the integration of them is carried out smoothly. Inside OROCOS, the inter-component communication is an important part of the whole framework. Communication patterns form a framework in itself, within the OROCOS framework. They assist the Component Developer, the Application Builder and the End User in building and using distributed components in such a way that the semantics of the interface is predefined by the patterns, irrespective of where they are applied (see Figure 2.4). OROCOS is rooted in SmartSoft [Schlegel and Worz, 1999], a component approach for robotics software based on communication primitives as core of a robotics component model. Dynamic wiring of components at run-time is explicitly supported by a separate pattern which tightly interacts with the communication primitives. This makes the major difference to other approaches. The patterns are based on the "SmartSoft" framework. OROCOS::SmartSoft is the name of the CORBA based implementation of the communication patterns.

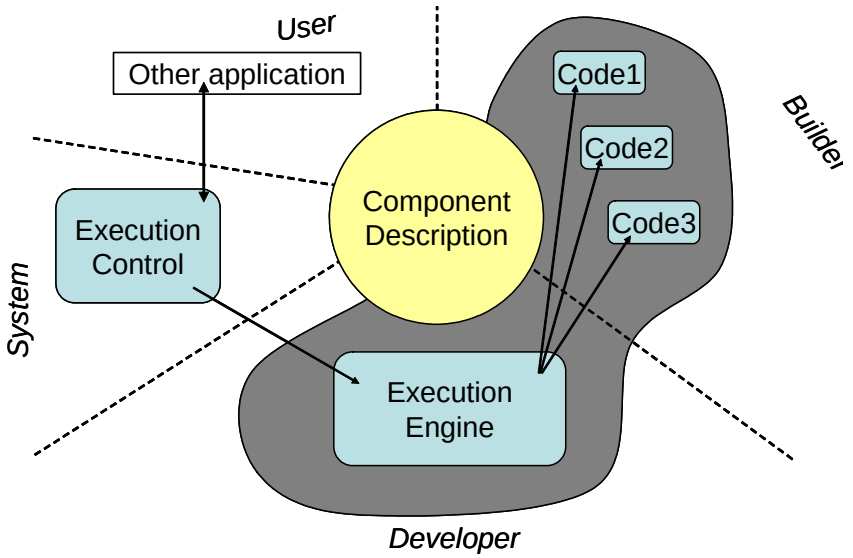


Figure 2.4: OROCOS software development model.

2.11 MOBILITY

One of the first commercial robotic systems using CORBA specification is provided by RWI (Real World Interface) [RWI, 1999]. The company sells a set of mobile robots using the CORBA 2.0 specification. The interesting part is the way it creates the access to the different robotic component of the system. They develop MOBILITY as its main middleware component. With this component, users from different locations can access the robot functions through a wireless communication scheme. All of the robots have the same interface but the number of components varies according to the model.

2.12 TELECARE ROBOTIC SYSTEM

Jia *et.al.*[Jia *et al.*, 2003] developed a basic telecare robotic system using CORBA specification. With this system it is possible to supply some basic services to aid the aged or disabled staying alone at home or small-scale distributed facilities. Main services are: Intelligent reminding, Data collection and surveillance or robotic services such as delivering of drug, juices or other things to the place where the the disabled or aged is.

2.13 TELEROBOTIC FRAMEWORK

Al-Mouhamed *et.al.*[Al-Mohamed *et al.*, 2003] developed a telerobotic system using the classic client-server approach. They proposed a reliable real-time connection between a master(client) and a slave(server) using Distributed Components (*.NET Remoting*). These components communicate each other using SOAP (*Simple Object Access Protocol*) and *.NET Remoting*. *.NET Remoting* is provided by Microsoft, and it represents an evolving step rooted on DCOM (*Distributed Component Object Model*).

2.14 Other middleware software

In our work we selected CORBA as the middleware component. This standard is rooted in previous models coming from the main software companies.

1. Microsoft's Distributed Component Object Model (DCOM) [Corporation, 1999]. This is a service that lets remote objects be treated as if they were local. DCOM transparently handles communication between atomic components, which are constructed from COM (Component Object Model) objects.

2. SUN Microsystems' Java Remote Invocation Method (RMI), for java applications.

3. Remote Procedure Call (RPC), one of the first distributed techniques for making "transparent" function calls in procedural modules.

Table 2.1: Characteristics of the different distributed robotic systems

System	Orientation type	Communication type	Object/Data oriented	Robot type	Middleware
TCA	Tasks	Centralized & Internal	Data	Mobile	NA
GLAIR	Agents	Internal	Data	Mobile	NA
CIRCA	Functions	NA	Data	Mobile	NA
TelRIP	Modules	Unix-socket	Data	Mobile	NA
ISAC	Agents	distributed	Data	Arm	NA
AuRA	Behaviors	Internal	Data	Mobile	NA
ARCO	Components	Distributed	Object	Mobile	CORBA
CAMPOUT	Components	Unix-socket	Data	Mobile	NA
MIRO	Functional Layers	Distributed	Object	Mobile	CORBA
OROCOS	Components	Distributed	Object	Mobile	CORBA
MOBILITY	Components	Distributed	Object	Mobile	CORBA
TELECARE	Services	Distributed	Object	Mobile	CORBA
TELEROBOTIC FRAMEWORK	Components	Distributed	Object	Arm	.NET

2.15 Summary

A chronological list of distributed robotic systems has been presented. The manner in how the authors divided the system varies from agents, components, functional entities, layers and behaviors or a mixture of them. Most of the aforementioned architectures tackled the issue of *Intelligent Behavior* by integrating these "smart" agents, components or functions into the design of the system. These agents, components or functional entities must operate on parallel or concurrently in order to achieve hard real-time constraints. Although most of the systems accomplished the goal or goals for what they were created, it is observed that the duplication of these systems is not an easy task. Furthermore, it is also observed how the use of middleware software is taking place in order to overcome the communication problem between these components or entities that are located in different platforms or use different operating systems. In the other hand, it is observed that most of the systems are oriented to mobile robots and a few ones focused on arm manipulators. In this work we are more oriented to arm manipulators.

Most of the aforementioned architectures provide a set of agents, or pseudo-agents to accomplish specific tasks or goals. In their minimum expression, they are called "atomic agent". All of the above works use different approaches of the agent concept and use a mixture of different relationships. Next, based on the platform, they select the communication media among these agents. As it is observed more and more systems are designed using a middleware software as its communication media. But using this middleware software "as it is" doesn't mean that the different entities necessarily can be reused on other projects or systems. For this reason, our approach in this work is focused on developing "building blocks" that can be used to create more complex systems, possibly agent-based systems. To develop these blocks an integration methodology is proposed. In this sense, we leave the door open to construct not just intelligent, but also complex systems for specific application areas.

To clarify the characteristics of a Distributed Robotic System and the challenges to face we provided a definition of these characteristics in the next chapter.

Chapter 3

Distributed Robotic Systems

3.1 Introduction

Since the first years of the computer era, the topic of distributed systems has been under intensive research. With the advent of the Internet, better communication equipment and faster and more capable computer systems, the possibilities for using distributed equipment put the computer science on another scale of evolution. However, the main advantages and challenges stay the same. A natural distributed system is a set of multiple, possibly different, robotic systems. This type of system has certain characteristics that make it very special. This chapter offers a review of distributed robotic systems (DRS) and the challenges we have to overcome to offer a better solution for the end user or for a specific task.

What is a distributed Robotic System (DRS)?
A general distributed system is a collection of *independent* computers or CPUs that appear to the users of the system as a single computer or a single system. Examples of this type of collection are:

a) a network of workstations allocated to users,

b) a pool of processors in the machine room allocated dynamically

c) a set of robots in a manufacturing cell.

d) a set of space exploration robots.

From here, we have that a distributed robotic system is a subset of a general distributed system. Also, we can define a distributed robotic system as a single robot controller that

interacts with other distributed components, such a vision system. In this case, there is a single user interface but there are several embedded system behind scene. Figure 3.1 shows this idea.

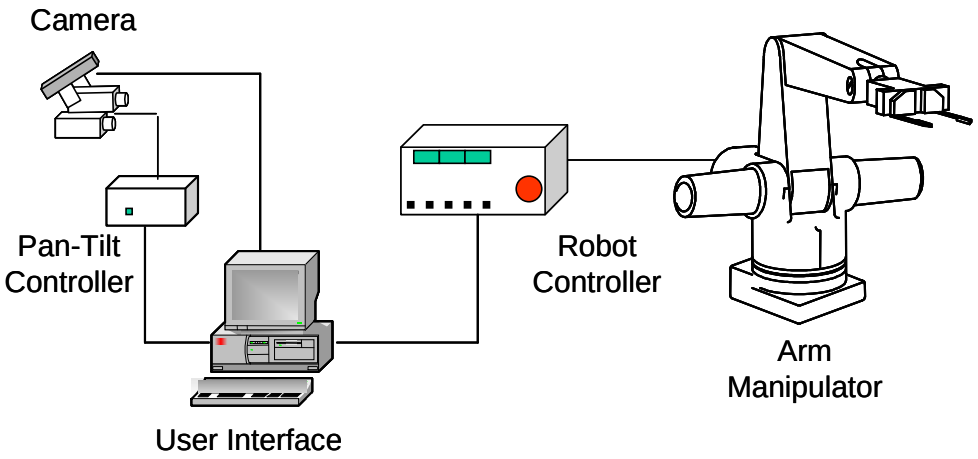


Figure 3.1: Distributed Robotic Components

Why Distributed?
Having several robots and components scattered through a laboratory, or distributed in several networks, we can achieve some benefits such as:

- Economics: Microprocessors offer a better price/ performance than mainframes.
- Speed: A distributed system may have more total computing power than a mainframe.
- Inherent distribution: Some applications involve spatially separated machines, such as a set of robots in a manufacturing cell.
- Reliability: If one machine crashes, the system as a whole can still survive.
- Incremental growth: Computing power can be added in small increments.

3.2 Main Features

Distributed robotic systems show many features, and among the most important we have the following:

a) Concurrent Execution: Multiple robots can process information in parallel.

b) Independency: Each robot works in its own Operating System and with its own resources.

c) Failure Management: The whole system doesn't fail if a robot computer crash.

d) Communication: The robots must have the ability to communicate with each other and they could be or could not be synchronized (no global clock).

e) "Virtual" robot system: The end user is not concerned with the number of robotic systems working for him. This is called *Transparency*.

There are several types of Transparency.

- **Access:** Local and remote resources are accessed using identical operations.
- **Location:** Resources are accessed without knowledge of their location.
- **Concurrency:** Several processes operate concurrently using shared resources without interference among them.
- **Failure:** Several minor faults can be hidden from the users.
- **Mobility:** The movement of resources and clients within a system (also called migration transparency).
- **Performance:** The system can be reconfigured to improve performance.
- **Scaling:** The system and applications can expand in scale without change to the system structure or the application algorithms.

3.3 Challenges

Although distributed robotic systems offer a set of very desirable features, this is not achieved without cost or difficulty. There are several challenges that the researchers must face to create a reliable and useful distributed robotic system. Table 3.1 shows these challenges. Furthermore, the integration of Distributed Robotic Systems is a very challenging problem. In universities, laboratories and companies we can find:

- many types of robots (SCARA, spherical, cartesian),
- many vendors,
- several team projects with different budget tackling similar problems,

- proprietary closed operating systems,
- proprietary protocols for monitoring and data file transfer,
- heterogeneous accessories, such as servo-grippers, vision subsystems and pan-tilt units.

An example of previous description is given with the equipment found in the Pattern Analysis and Machine Intelligence Laboratory (PAMI-Lab) at the University of Waterloo. This is shown in Figure 3.2

Table 3.1: Challenges to face when creating a distributed robotic system

Aspect	Challenge Description
Heterogeneity	Are different network types, hardware, and operating systems compatible?
Openness	Can you extend and re-implement the system?
Scalability	Can the system behave properly if the number of "robots" and components increase?
Failure Management	How does the system responses to partial robot or component failures?
Concurrency	When there are multiple robots or subsystems sharing resources, how does the system maintain the integrity of the resources and a properly operation without interference?

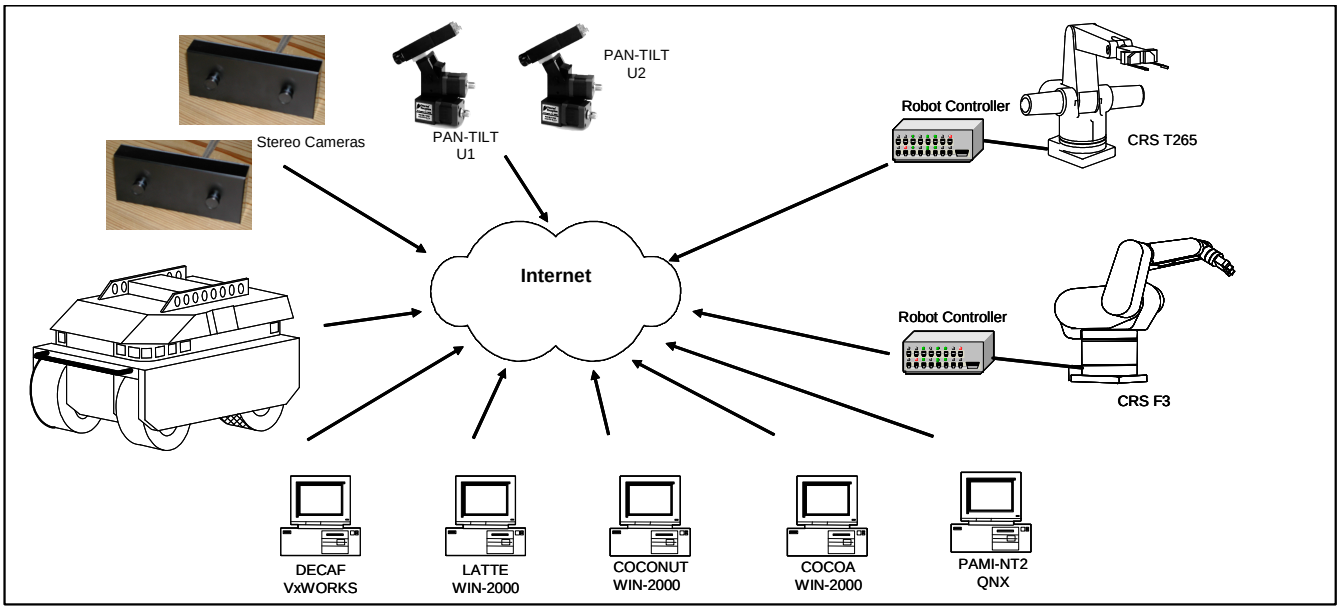


Figure 3.2: Robots and accessories at PAMI-Lab at University of Waterloo

3.4 Robot Networks

A DRS is based mainly on a robotic networks with some subsystems distributed in it. Given two components or robots located possibly in the same robotic network or in different networks, the following questions come up:

- How are the request and response transmitted between the requestor and the server?
- How do the request and reply messages move through the network?
- What are the modes of communication between the requestor and the responder?

Similar to human behavior, robotic networks have a set of communication protocols to facilitate the transfer and reception of messages among computers. A protocol defines the format and the order of messages sent and received among network entities, and the actions taken on message transmission and receipt. In the machines all communication activity is governed by *protocols*. In this sense, the International Organization for Standardization (ISO for its Greek root) creates the Open Systems Interconnection (OSI) layered protocols for computer networks.

3.4.1 Layered Protocols

The OSI specification divides functionality into different layers and allows each layer to provide one function. Table 3.2 shows the main functions and some examples of well known implementations for each layer.

3.5 CORBA specification

CORBA specification is a set of distributed systems standards promoted by the Object Management Group (OMG) [OMG, 2000]. The idea behind CORBA is to allow applications to communicate one with another no matter where they are or who has designed them. The basic idea is to create an interface that can be used or understood by every application on different equipment. To achieve this goal an *Interface Definition Language* (IDL) is created. The CORBA specification follows a Client/Server approach. Usually the Server describes its services through this interface. Every application that needs to share its executable code must create an IDL file to be distributed over the network (see Appendix A for more information).

Table 3.2: OSI Protocol Summary

Layer	Description	Examples
Application	Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service.	HTTP, FTP SMTP, CORBA IIOP
Presentation	Protocols at this level transmit data in a network representation that is independent of the representations used in individual computers (which may differ). Encryption is also performed in this layer, if required.	Secure Sockets (SSL) CORBA Data Rep.
Session	At this level reliability and adaptation are performed,	
Transport	This is the lowest level at which messages (rather than packets) are handled. Messages are addressed to communication ports attached to processes, Protocols in this layer may be connection-oriented or connectionless.	TCP, UDP
Network	Transfers data packets between computers in a specific network. In a WAN or an inter-network this involves the generation of a route passing through routers. In a single LAN no routing is required.	IP, ATM virtual circuits
Data link	Responsible for transmission of packets between nodes that are directly connected by a physical link. In a WAN transmission is between pairs of routers or between routers and hosts. In a LAN, it is between any pair of hosts.	Ethernet MAC, PPP ATM cell transfer
Physical	The circuits and hardware that drive the network. They transmit sequences of binary data by analog signalling, using amplitude or frequency modulation of electrical signals (on cable circuits), light signals (on fibre optic circuits) or other electromagnetic signals (on radio and microwave circuits).	Ethernet base-band signalling, ISDN

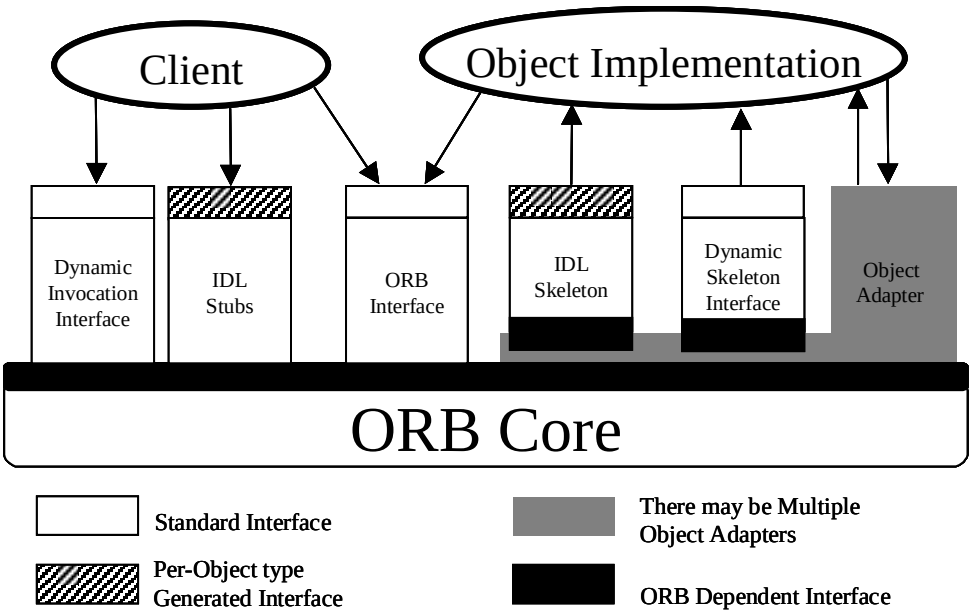


Figure 3.3: Object Request Broker (ORB) interface

CORBA is based on an *Object Request Broker* (ORB), a mechanism through which distributed software and their clients may interact. It specifies an extensive set of bus-related services for creating and deleting objects, accessing them by name, storing them in persistent store, externalizing their states, and defining *ad-hoc* relationships between them. This module takes care of the interfaces and makes the changes needed to transmit and to marshal data, as is shown in Figure 3.3. CORBA specification provides a method of creating interfaces between equipments to facilitate their communication. It is based also on an object-oriented design and implementation.

3.6 Summary

Distributed Robotic Systems consist of autonomous robotic systems and components that work together to make the complete system functions as a single robotic system or a coordinated team. They offer a set of good features that permit the increase of power processing at low cost. In order to provide a good performance many challenges must be overcome to create a reliable system. The challenges are based on the number of operating systems, communication aspects, transparency and management of failures. Normally, robotic systems come with their own customized or proprietary system, so *integration* of any other component becomes a difficult task. In the past, researchers

created their own interfaces to connect each equipment, and still it is a normal practice. Most of the aforementioned architectures provide a set of agents, or pseudo-agents to accomplish specific tasks or goals. In their minimum expression, they are called “atomic agent” or “actors”. All of the above works use different approaches of the agent concept. Next, based on the platform, they select the communication media among these agents. This is not our case. Our approach is to focus on developing “building blocks” that can be used to create more complex systems, possibly agent-based systems. In this sense, we leave the door open to construct not just intelligent, but also complex systems for specific application areas. Nowadays there are several middleware technologies that alleviates the burden of communication issues to solve. These middleware softwares appear on the top levels of the OSI standards. CORBA specification is one of them and its main structures and functionality was briefly described. But CORBA doesn’t build the application itself, it is just a middleware that alleviates many problems related to communication issues. In order to create software modules that can be easily integrated it is required to apply very good design practices from the very beginning of the development cycle. In the other hand, in the robotics area there are many legacy, proprietary, and closed systems that need to be integrated in order to work as a distributed robotic team. Then, a methodology that deals with this kind of problems and that uses CORBA as the communication bus is proposed in the next chapter.

Chapter 4

Methodology for Integration of Distributed Wrapper Components

The component-based development approach is becoming more and more popular for creating Internet-based applications [Mecella and Pernici, 2001]. In this chapter, a methodology based on wrapper components is presented as a modular approach in order to create building blocks of complex applications. Distributed Robotic Systems by using legacy systems or new object oriented modules are some examples. Their characteristics and applicability are discussed in this chapter.

4.1 Definitions

In order to have a consistent understanding of the methodology some definitions adapted from [Mecella and Pernici, 2001] and [D'Souza and Wills, 1998] are needed.

Component: A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger. For one component to replace another, the replacement component need not work the same internally. However, the replacement component must provide at least the service that the environment expects of the original and must expect no more than the services the environment provides the original. The replacement must exhibit the same external behavior, including quality requirements such as performance and resource consumption.

Provided and required interfaces: A component is specified explicitly through the definition of its interfaces. A *provided interface* describes what the client can expect from the component. In the other hand a *required interface* describes

what the component expects from the environment. The interaction among the components is only established through these interfaces.

Abstraction Level: Different models support different abstraction levels. One way to measure this concept is through the semantic and technological coupling. For low abstraction levels corresponds a closer (tight) coupling. For example, Figure 4.1 shows how the position information of a robot can be accessed. Mainly the access can be done:

1. Through accessing a register in the robot controller.
2. Through a remote procedure call (e.g., Get(Location,...))
3. Through an object class "robot" expressed in an IDL middleware.

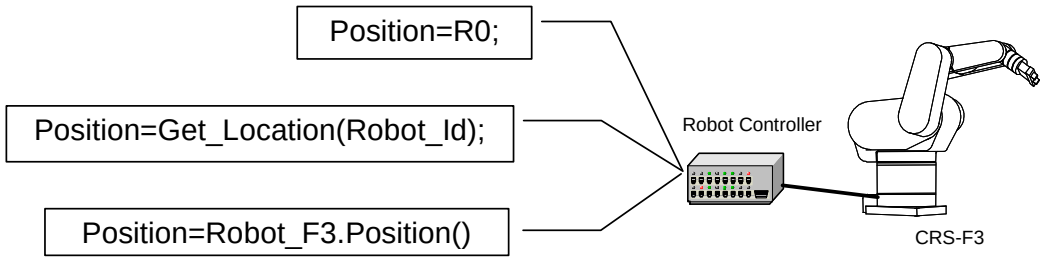


Figure 4.1: Different ways of accessing or getting a robot position.

For a component to plug-replaceable, it is essential that the component specifications be self-contained and symmetrical; in this way we can design reliably using parts without knowledge of their implementations. The abstraction levels shown in Figure 4.1 are different. The first one shows how the information is physically registered in the host system, meanwhile the last scheme facilitates the access to the information but it doesn't show details about how the information is internally managed. Component software demands a complete separation of interface specifications from their implementations. The interface specs, rather than the source code, define what a component will provide and expect when used.

Interfaces developments made *ad-hoc* for a specific application is costly, difficult to maintain and it limits the scope of the application to a few cases. Due to this, two modeling schemes based on object technologies are defined. The first one is based on the *operation* and the second is based on the *concept*.

4.1.1 Component definition oriented to operation vs to concept

In this section definitions of these two types of components are discussed, based mainly on the *state* of the object. Before this some basic definitions are given:

Association property: The component specification is composed of attributes/properties. These *properties/attributes* are considered as part of the *object state* in a component instance if it is necessary to keep its value between two any invocations of its *services/methods* in such component instance.

Completed State Component: The specification of this component provides the state association. The management of the component's state is carried out by the *server* in a distributed application.

Stateless component: In this case, any of the following situations can occur:

a) The component specification provides the associative property but the management of the component's state is carried out by the *client* in a distributed application.

b) The component specification does not provide the associative property. It is a pure *function*.

Based on this, ***Concept-Based components*** are defined as ones whose objects are a representation of the real world. These are *completed state* components. On the other hand, ***Operation-Based components*** are ones without a direct association with the real world, but whose operations in objects of the real world are modelled. These are *stateless* components.

Examples of previous definitions are:

a) A robot object in which methods to read its position and to command it are present. The commands change the state of the robot. This is a *concept-based* component. Last option on Figure 4.11 outline this type of component.

b) The other example is a generic operation, which can be applied to similar objects. The operation needs the object reference and pertinent data to execute the operation. There is a return code indicating if the operation was successful or failed, or in case of query the information requested is received. This last example is very common in many of the legacy systems, so the *operation-based* component can be visualized as access components. Second option of Figure 4.11 represents this type of component.

4.2 Application Development Based in Components

A component can be defined as a "design unit (at any level), for which the internal structure is defined. It has a name associated with it and there are some design guidelines to integrate this component and to illustrate how it can be reused". This is a general definition but we can describe it in more specific detail:

Conceptual Component: It is a model/scheme (or subset) that can be reused, following an object modeling approach. This could be specified with the Unified Modeling Language (UML) [Rumbaugh *et al.*, 2000].

Software Component: a coherent software package that can be developed and delivered independently. It has explicit and well defined interfaces for the services that it provides and for the expected services from others components. This kind of component can be composed of subcomponents but without modifying the external interface. The term instance of a component is used to distinguish the specification from its executable. In an object oriented approach, a component is a set of classes assembled together to be delivered as a simple software unit.

An aspect of modelling components is the *granularity* to which the components are defined. The trend is to have more fine components. Prior to this, a subsystem was considered as component in a bigger system. With this new trend, we have smaller components but with a better utilization for various applications. For example, a component to sort any kind of data, or a component to create graphs in any environment is now possible.

The biggest obstacle for the component-based development is the necessity of a common framework, i.e. a definition of the "world where the component will live". Recently, new emerging technologies based on internet, provide the backbone for the effective development of components. These developments are based on distributed computing technologies, on middleware software and they are object oriented, specifically in the message transmission between these objects. CORBA specification is one of these technologies, and we selected as our development platform.

4.2.1 Integrating legacy systems with distributed applications

Legacy systems are defined as applications with a critical value which have already been in production. According to this definition most of the current systems are of this type. The proposed re-engineering strategies to deal with these systems are the following:

a) Integrate

b) Migrate

In the first option, the old systems are integrated into new applications creating new interfaces. In the second case, the old systems are replaced progressively until a totally new software application is achieved.

The interfaces generated for the old systems are denominated ***wrapper interfaces***, and we have two types:

Access Interface, in which there is only a mapping between the data and the path access to them, and

Integration Interface, in which new interfaces are generated and they provide a higher level of abstraction. Also, these interfaces can have a subset of access interfaces.

The following Table 4.1 shows the characteristics to be taken into account during the development of systems based in the above components.

Table 4.1: Development characteristics of component-based systems through legacy systems [Mecella and Pernici, 2001]

Characteristic	**Concept-Based**	**Operation-based**
Wrapper type	Integration	Access
Design Complexity	High	Low
Development time	Long	Short
Integration Logic	Distributed	Centralized
Application Composition	Easy	Difficult
Application development time	short	long
Number of active instances	High	Low

In systems developed with concept-oriented components, there could be a big number of low level layers that affect the overall performance. CORBA has this problem, and the

different software developers focus their marketing effort to differentiate their products. In our case, in robotic industrial applications a Real-Time CORBA [OMG, 2000] is proposed.

4.3 Proposed Methodology

Our methodology follows three basic steps, as is shown in Figure 4.2:

1. Dividing
2. Wrapping (Encapsulating)
3. Connecting

These steps are related in such way that the definition or results of each one affects the performance and scope of the other. Furthermore, due to the fact that we will use CORBA as a middleware in order to connect different modules, some considerations must be taken into account.

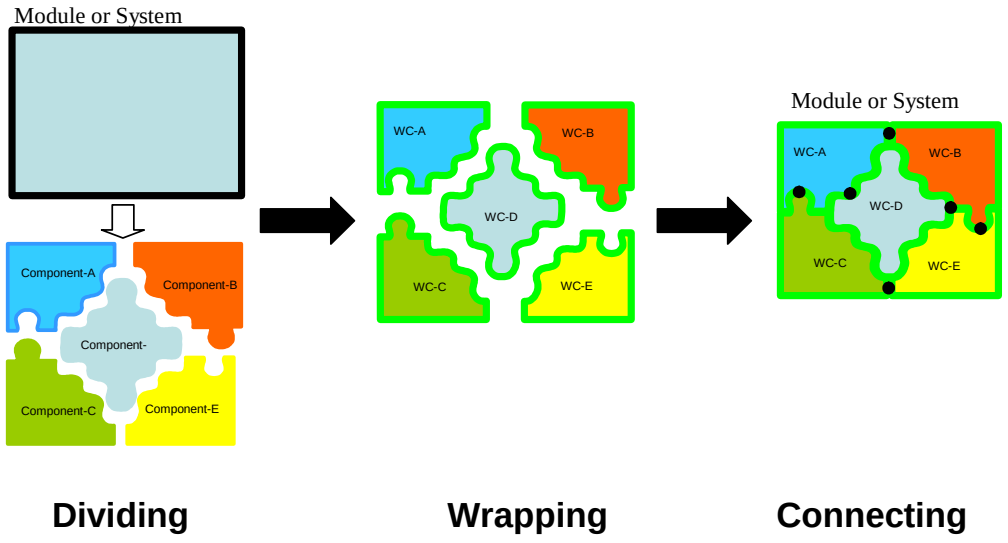


Figure 4.2: Main steps followed when defining wrapper components

4.3.1 Dividing

There are several considerations when dividing a system. We will mention the properties that we consider a key issue for a better design of a component. Dividing a system into

small subsystems is an old practice to deal with big projects. In many cases a common sense policy is followed or a top-down strategy is applied in order to have a well separated and defined modules. In this methodology we propose some indexes for a set of criteria in order to have a better understanding of the dividing process. At the end of this stage is expected to have a table with a number of well defined modules.

Physical constraints

Diving a system or application could be done in a conceptual fashion, but in most of the cases we will find a physical partition at different levels. For instance we will need to integrate *Commercial-off-the-Shelf* (COTS) software or legacy systems that use specific hardware, libraries or platform. In some cases, there are subsystems that cannot share limited resources, such as communication ports or memory. In this sense, the parts in a system could coexist in the same hardware (PC or main-frame) or in a distributed manner using different operating systems or platforms. Then a system designer must deal with all kinds of software components instead of doing these components from scratch. Figure 4.3 shows some examples of physical constraints. Usually, each component has its own CPU, its own communication interface and its own operating system. In many cases they must communicate each other through a CPU in the middle.

The index for this type of division is named as

$N_{pc} = Number\ of\ components\ separated\ by\ physical\ constraints$

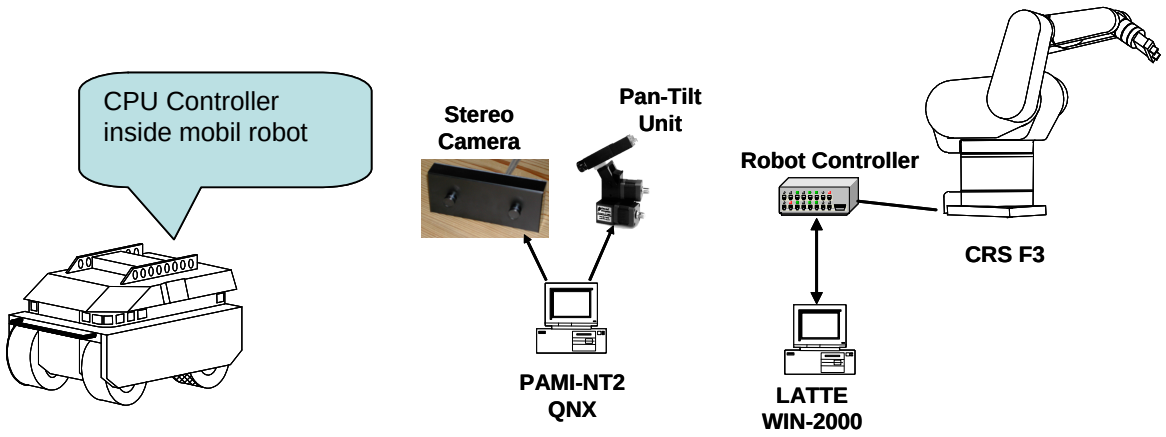


Figure 4.3: Physical constraints of the components needed to create a robotic system.

Performance

Another consideration to split a system is the requested performance (i.e response time) to achieve certain goals. In this case, we are talking about the CPU usage when multiple tasks are running in the same single CPU. This could be the case when the system cannot respond on time because of internal delays or worst, because of the internal blockage of resources. Then it will be necessary to put the components in different CPUs or use a better CPU scheduling algorithm, Figure 4.4 depicts this idea.

The index for this type of division is named as

N_p = *Number of components separated by performance issues*

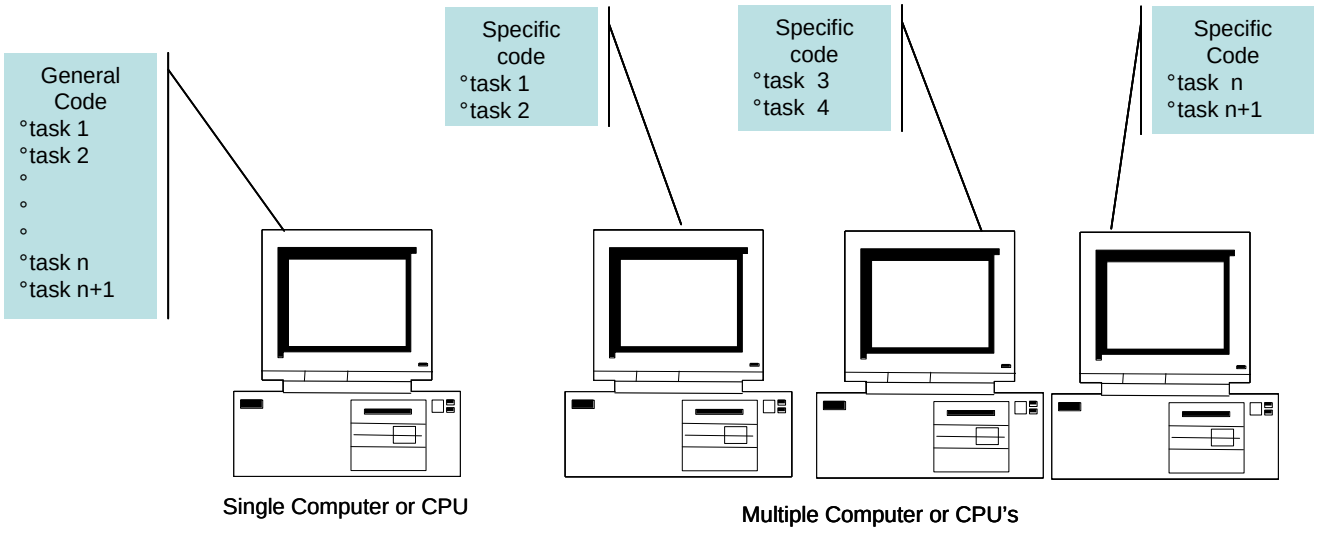


Figure 4.4: Splitting of a task into several parallel and concurrent tasks to enhance system's performance.

Abstraction

The development of a system is a task where the main parts of such system must be defined. This definition process provides an outline of the functions and the attributes of these parts. Next, each part is subsequently divided into smaller subsystems with their own functions and attributes. Functions and attributes of a child subsystem could be a subset of the parent system. This partition activity can continue in the different subsystems until certain level is achieved. During this process we may find that some subsystems could have similar functions and attributes. Figure 4.5 shows this idea. Functions F1-a, F1-b and F1-c are similar. In this case we can make an abstraction from the main functions for the set of subsystems. This abstraction can be parame-

terized in such way that with minimum changes the abstract function can supply the functionality requested by every subsystem. The abstraction process can be applied to classify different types of robot. For instance, it is possible to have *mobile* robots and *arm manipulators*. Each type of robot would have a specific instruction or command set, that can be applied to all robots pertained to the same type.

The index for this type of division is named as

N_f = *Number of components separated by functional abstraction*

Dividing a process by functional abstraction creates components at different levels. In our case we only distinguish two levels: *Level-1*, it refers to all components or subsystems identified as primary elements to create a specific system; *Level-2*, it refers to all subcomponents that compound a bigger component or a subcomponent that can be used in several components. In the last case this subcomponent can be referred as a *generic component* (GC). In Figure 4.5 F2, F3 and F4 are just simple subcomponents, meanwhile F1-x is a generic subcomponent.

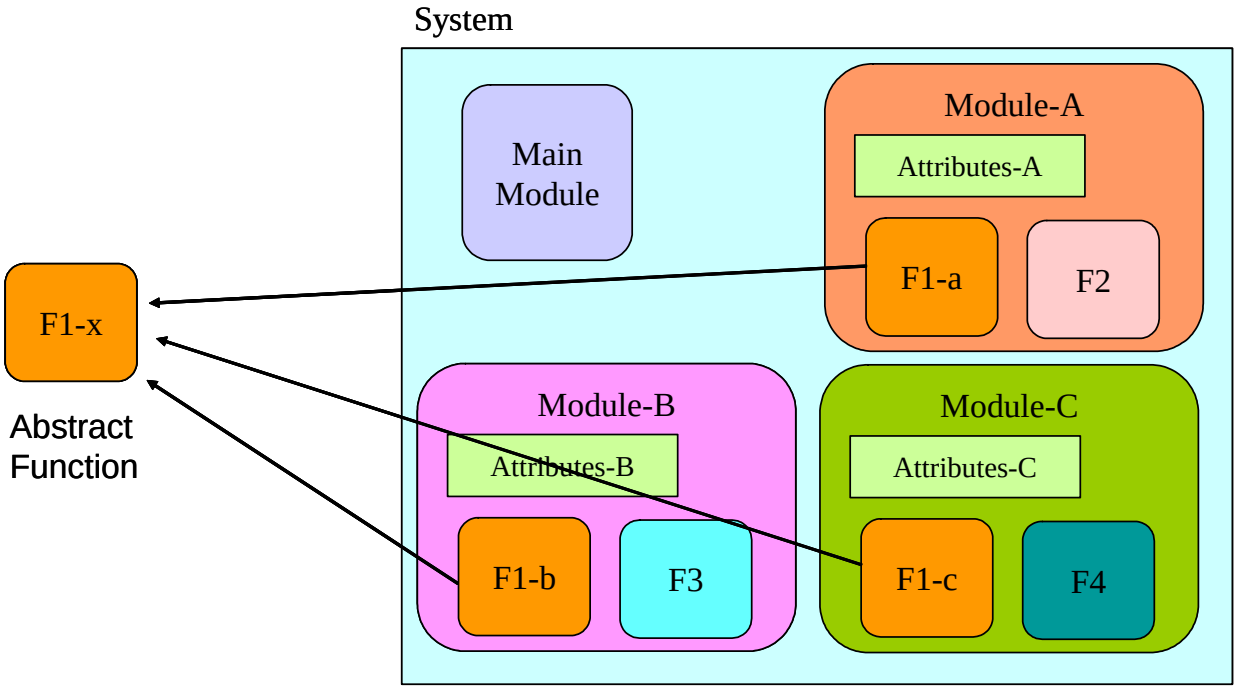


Figure 4.5: Abstract functionality of similar parts in system construction.

Interconnections

Dividing a system into subsystems generates smaller and bounded components that are easier to manage and to build, but at the same time the number of interconnections

among these components could grow rapidly (see figure 4.6). This situation could create communication problems when we have one-to-many relationships, i.e. the management of open-close sessions of each connection and its continuous checking for abnormal ending. Here we can take advantage of one of the services provided by CORBA, specifically Event Service. This service is explained in detail in appendix A, Section 3.2.

The index used for this case is related to each component

N_{c_i} = *Number of connections of the component i*

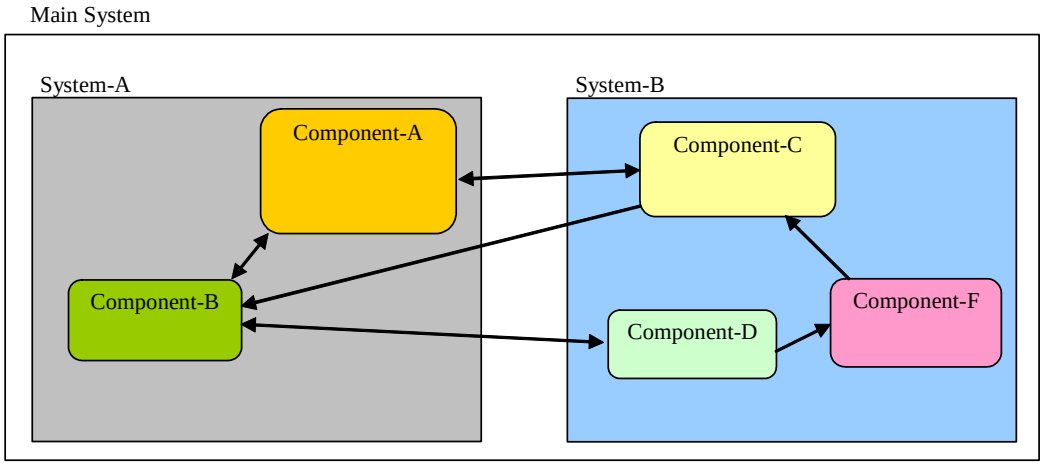


Figure 4.6: Interconnections between components and subsystems

The dividing process can be checked using the flow diagram shown in figure 4.7. The first considerations are the physical constraints, followed by the performance requested. The last two considerations are related to some extent. For example, creating many functional abstraction will lead to an increment on interconnections, in the other hand if we keep a small number of interconnections the component of the system will connect easier but the number of exchange modules will be reduced. This means that a small subcomponent is easier to exchange than a bigger subcomponent.

With this reduced number of counters it is possible to compute the following indexes, where N_{wc_1} is the total number of wrapper components at level-1, N_{wc_2} is the total number of wrapper components at level-2, and N_{wc} is the total number of wrapper components at any level The wrapper component concept is explained in the next section of this chapter.

Complexity Index

$$I_{cpx} = \frac{\sum N_{wc_1}}{\sum N_{wc_1} + \sum N_{wc_2}} = \frac{\sum N_{wc_1}}{N_{wc}} \tag{4.1}$$

If I_{cpx} is equal to 1, that means that only *Level-1* components have been defined for the system, reducing the complexity but at the same time reducing the possibility of exchange subcomponents.

Structural Index

$$I_{struct} = \frac{N_{pc} + N_p + N_f}{N_{wc}} \tag{4.2}$$

If I_{struct} is equal to 1, it means that each component only have one criteria for its partition. On the other hand if the I_{struct} is close to 3, then it means that each component applies for all partition criteria.

Connection Index

$$I_{cnx} = \frac{\sum N_{c_i}}{N_{wc} * (N_{wc} - 1)} \tag{4.3}$$

If each component connects will all $N_{wc} - 1$ components, then the index I_{cnx} has a value of 1. If each component connects only with one component, then the index I_{cnx} will have a value of $1/(N_{wc} - 1)$.

The information generated at the end of this stage is shown in the Table 4.2

Table 4.2: Outcomes of the dividing process, these include the counting for the different indexes

Component Name	**Component level (1,2)**	**Physical Constraint**	**Performance**	**Abstract Functionality**	**Connections**
Component-1	Level-1	$\surd$	NA	NA	N_{c_1}
Component-2	Level-1	NA	$\surd$	NA	N_{c_2}
Component-3	Level-2	NA	NA	$\surd$	N_{c_3}
Component-4	Level-1	NA	$\surd$	$\surd$	N_{c_4}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Component-n	Level-2	$\surd$	$\surd$	$\surd$	N_{c_n}
Total Number of Components	Total Components for each level	Total Physical Components	Total Performance Components	Total Abstraction Components	Total Number of connections
$N_{wc} = \sum N_{wc_i}$	N_{wc_i}	N_{pc}	N_p	N_f	$N_c = \sum N_{c_i}$

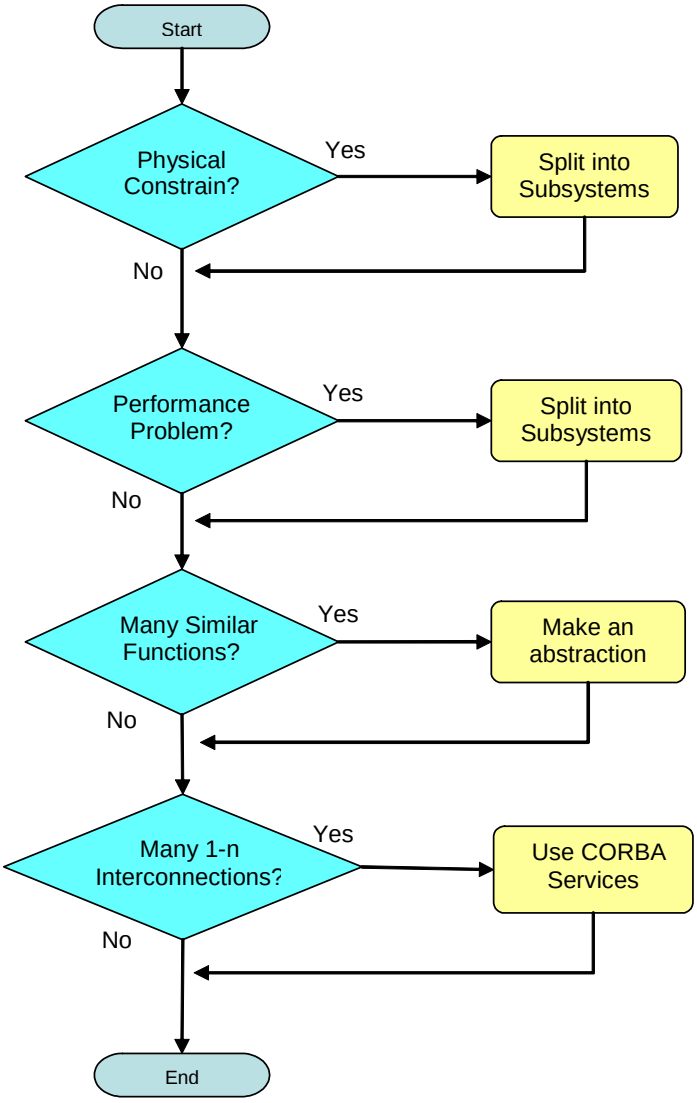


Figure 4.7: Flow diagram to divide a system into several components.

4.3.2 Wrapping (Encapsulating)

As the name suggests, a Wrapper Component (WC) is a programming module that encapsulates an abstract functionality for specific hardware or software modules in the system. There are different models depending on the abstraction level: *Conceptual Approach or Operational Approach* as we explained in section 4.1.1. In our methodology, the basic configuration of a Wrapper Component consists of three main blocks:

- IDL (Interface Definition Language) interface
- Transformation/Interpreter Code, and
- Hardware/Software object/Library Implementation.

Figure 4.8 shows these blocks, where a robot is considered for the hardware/software object implementation.

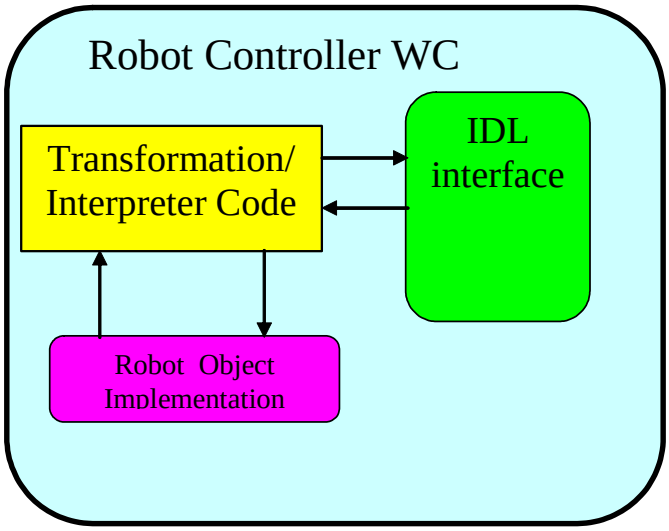


Figure 4.8: Basic elements in a Wrapper Component.

IDL Interface

This is an interface definition for a particular *class* of components; its actual definition is a key issue to construct reusable and easy to connect components. We define three basic functionalities for this interface: *abstraction, monitoring, and configuration.*

Abstraction: By *abstraction* we refer to the particular functions (methods) set that a specific component class must have without taking into account the implementation details. The abstraction level could start from an *access* level where there is

a mapping one to one with a legacy system, until a higher level, as it is shown in Figure 4.1. A small number of powerful services is preferred over a large number of single services. Furthermore the abstraction on the interface must facilitate the integration and management of the component as a replaceable unit. Figure 4.9 shows two robotic interfaces with a small number of methods or services. In option (a) the designer must know the right axis to move for a specific robot, and somehow it must look for the way to move several axes at the same time, meanwhile in option (b) the designer only needs to use a single method according to the movement to realize. Option (a) is more operation oriented while option (b) is conceptual oriented.

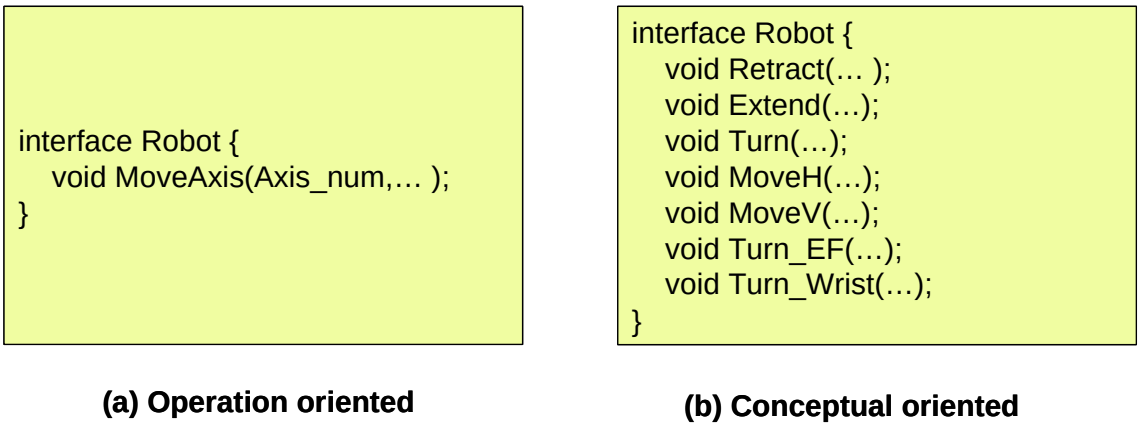


Figure 4.9: Two different abstract interfaces for moving a robot.

Monitoring: By *monitoring* we refer to the general functions that every component must have to query its internal states. This monitoring could be by request or depending on the application must be started and stopped through some configuration methods. Figure 4.10 shows an interface definition which includes this feature. A start/stop monitoring is explained when we develop the vision server, see Appendix C for details.

Configuration: Finally, *configuration* represents the capability of changing the internal attributes of the component according to external requirements. Usually, these changes will affect the behavior of the component. In the case of the robot interface, there is a method to learn a specific position or location. See Figure 4.10. In the case of a vision component, this could be the rate at which the images are captured or transmitted. Other configuration could be if the image is captured in color or monochromatic.

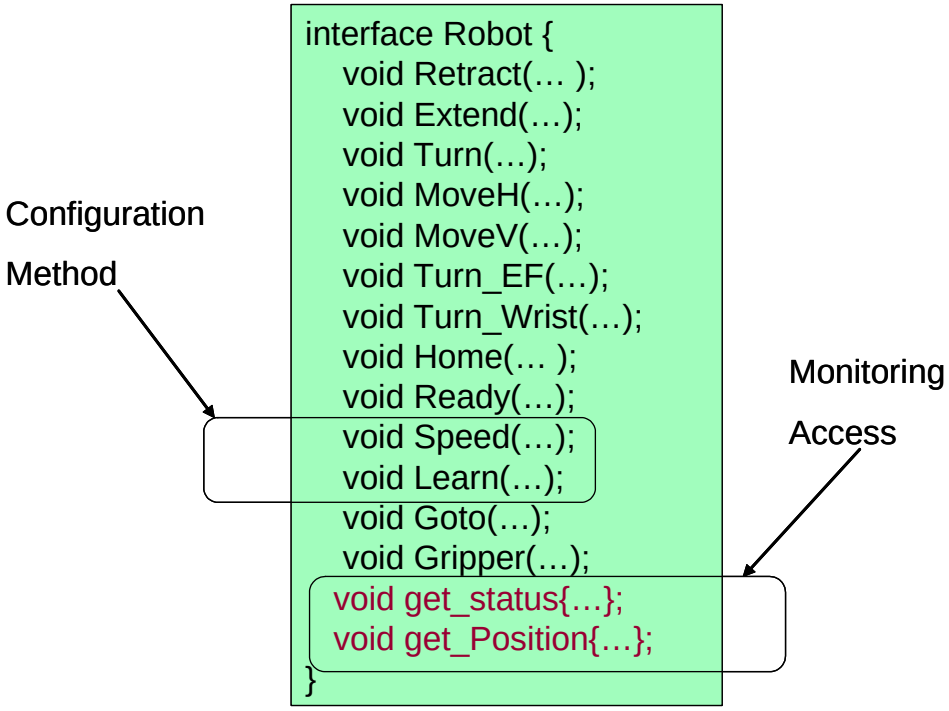


Figure 4.10: Monitoring and Configuration methods for accessing and configuring the information of a robot server.

Transformation/Interpreter Code

This element requires more effort compared to the other elements when implementing a wrapper component. Basically, in this part of the component, the *component builder* (people in charge of the development of this part) has to do the following basic tasks: *Data transformation, Data Interpretation, Data Distribution, Data Collection and Data Processing.*

Data Transformation: This corresponds to wrapping all differences on data definition between the interface and the object implementation. For example, if the object implementation manages the position of an axis using a long integer data type but the interface is declared as a float data type, the transformation code makes this change on the data type both ways. Figures 4.11 and 4.12 show this idea.

Data Interpretation: In many cases due to the different abstraction levels between the component's interface definition and the current legacy system there is not a matching one-to-one of the functions. Then an interpretation or translation is required in order to achieve the service offered in the interface definition. For example, if there is a function to move a robot by extending the arm and this function doesn't exist in the legacy robot API definition, then it is necessary to execute a set of primitive provided functions to achieve the required service. A

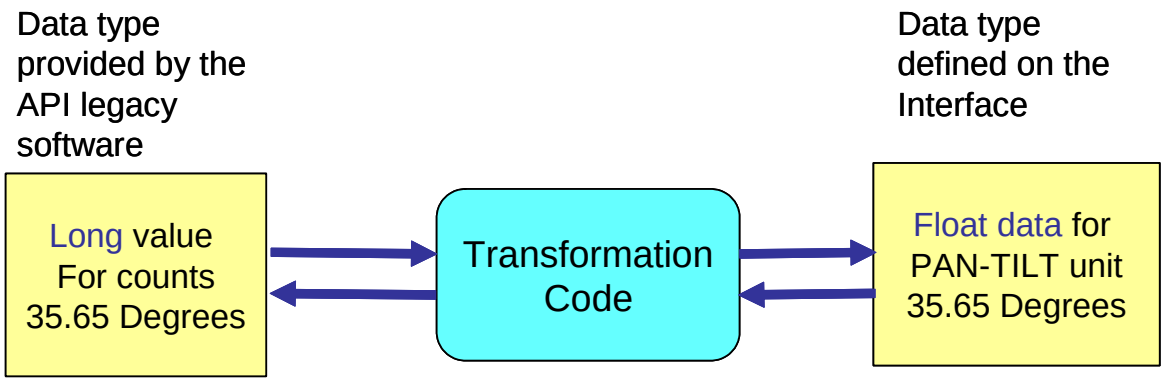


Figure 4.11: Data transformation task between the interface definition and the object implementations.

more detailed example of this case is explained in Appendix B. Other example is moving the wrist of the robot. Figure 4.12 depicts this idea. The wrist axis number is different for two different types of robots. The interpretation task selects the right axis according to the type of robot.

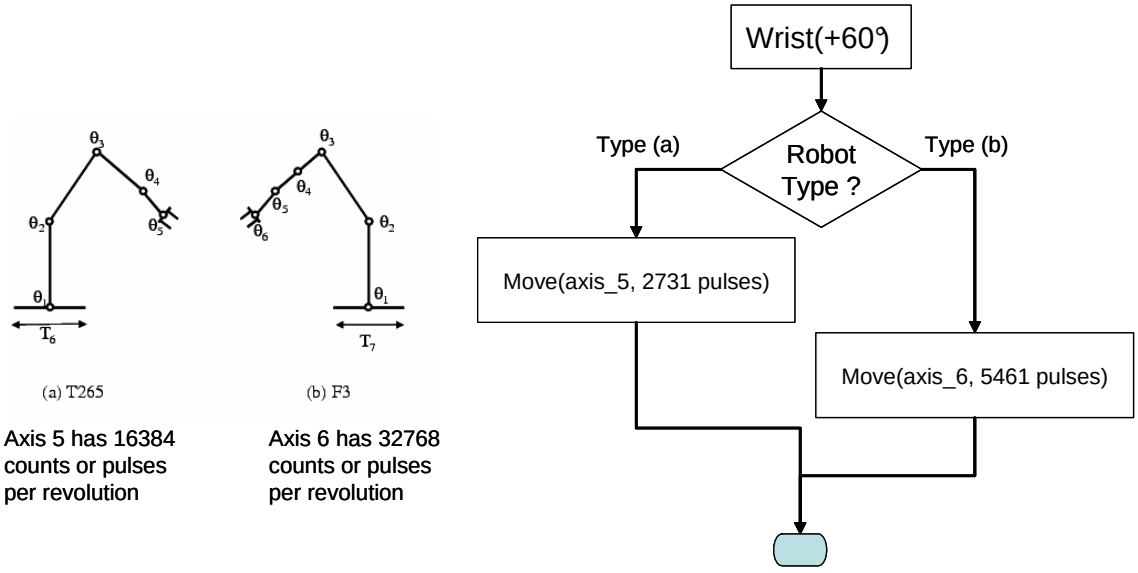


Figure 4.12: Data Interpretation task carried out to select the right axis when moving the wrist of an arm manipulator.

Data Distribution: This task refers to the data distribution among the different subsystems that compose a wrapper component. This is analog to the courier service provided in a corporative building. Normally a package with letters and documents arrives to a specific area or department. Next, the information is classified and separated according to the different areas on the building and smaller packages

are delivered. In a robotic system, it could be the case that different background process are running on parallel or concurrently and they need or supply different types of information. For example, a subsystem for managing a database of locations coexists with the motion controller subsystem. An abstract function such as LEARN(position) could request the current position to the motion controller and then pass this information to the database manager. Figure 4.13 depicts the general concept of this task. In some cases, the data distribution is forward while in more complex components requires extra processing.

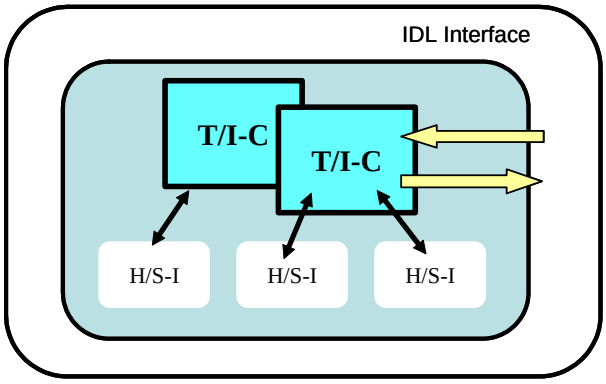


Figure 4.13: Data Distribution task carried out to delivery information to the different subsystems in a complex component. T/I-C stands for Transformation/Interpreter Code and H/S-I stands for Hardware/Software Implementation. Inside a Wrapper Component could coexist several H/S-I and T/I-C, but this is hidden to the external users.

Data Collection: This is the opposite task to the previous one. Here the data from the different subsystems is collected, sometimes formatted or arranged into data structures to reply the information through the interface.

Data Processing: Finally, data processing task corresponds to all computing steps used for formatting, data sorting, numerical calculation, parsing, concurrency management or any other computation tool used to process the information.

Hardware/Software Object implementation

Integration into the whole system is demanded for this element. Usually, this element is defined for a specific hardware or it is supplied by a specific vendor. In most of the cases, this component is provided "as is" with an Application Program Interface (API) definition. However, it is difficult for specific purposes, and sometimes impossible, to access the low level code. Examples of these elements could be robot motion or control libraries, image processing libraries or database libraries. Most of them are in binary

format and it is difficult to determine their code or disassemble them. Depending on the type of element, this could coexist in the same CPU where the Wrapper Component is launched, or it could be running in a different CPU with a communication link provided to be accessed by the Wrapper Component. Figure 4.14 shows these cases. Usually, option (b) is found in all elements or components that implies a hardware integration, meanwhile option (a) is found in software libraries of intangible elements, such as database libraries, image processing libraries, or math libraries, among others.

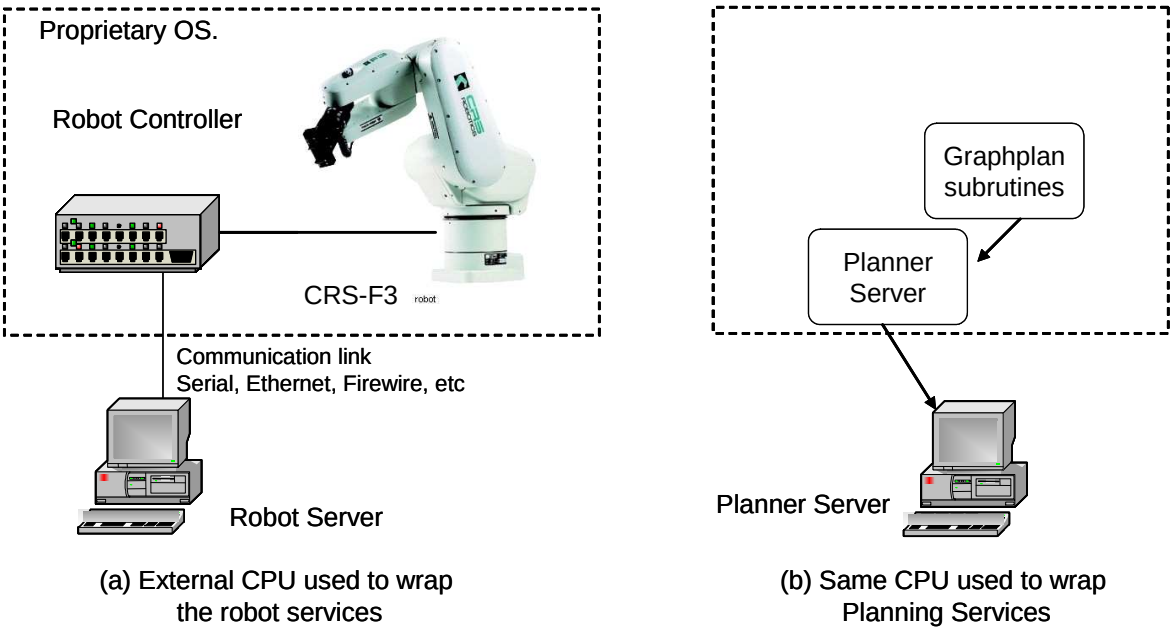


Figure 4.14: Hardware/Software Implementations and the different options of their locations. (a) The Robot Server Interface can not be located in the same CPU due to its proprietary and closed operating system, usually in this type of components a communication interface is provided by the vendor for a specific media. (b) Both the Planner Server and the planning subroutines are located in the same CPU. Usually in this type of component, the libraries and the wrapper component are developed under the same language, platform and operating system.

The outcomes expected for the Wrapping stage are summarized in Table 4.3. For each IDL interface definition we can have an abstraction level based on the relationship between the input functions of the component to wrap and the final number of output functions defined on the IDL interface.

Abstraction Index

$$I_{abstract} = \frac{Out - functions}{In - functions} \tag{4.4}$$

When this index is equal to 1, it means that the component is defined as an *Access*

Component. This is normal in components with a few number of functions or methods. When this index is below 1/2, it means that the component has some abstraction functionality and it wraps most of the main functions of the original component. Usually a low value in this index represents a big effort on the Transformation/Interpreter codification.

Table 4.3: Outcomes of the Wrapping process

Component Name	**IDL Interface Definition**	**Transformation/ Interpreter Code**	**Hardware/ Software Integration**
Component-1	√	√	Hardware
Component-2	√	√	Software
⋮	⋮	⋮	⋮
Component-n	√	√	Both

4.3.3 Connecting

Due to the aforementioned components were encapsulated using CORBA specification, the communication among them is supported transparently by the different ORB vendors. However, there are some aspects or characteristics in the connection that we can improve using different services from CORBA, namely Naming Services and Event Services.

CORBA is a software bus that integrates different types of operating systems and programming languages. It acts similar to a hardware bus that connects different interface boards. CORBA follows the Client-Server approach to connect two applications. As shown in figure 4.15, a client invokes an operation on the server side as if it is a local object. A client stub marshals a parameter, and the ORB (Object Request Broker) core locates and delivers the request to a server. A server skeleton demarshals the request, and an object adapter calls the function. After the operation finishes, the server skeleton marshals return values. The ORB core delivers the return values to the client, and the client stub demarshals it. Lastly, the client obtains return values from the server object. This type of connection is known as *Request/Reply*.

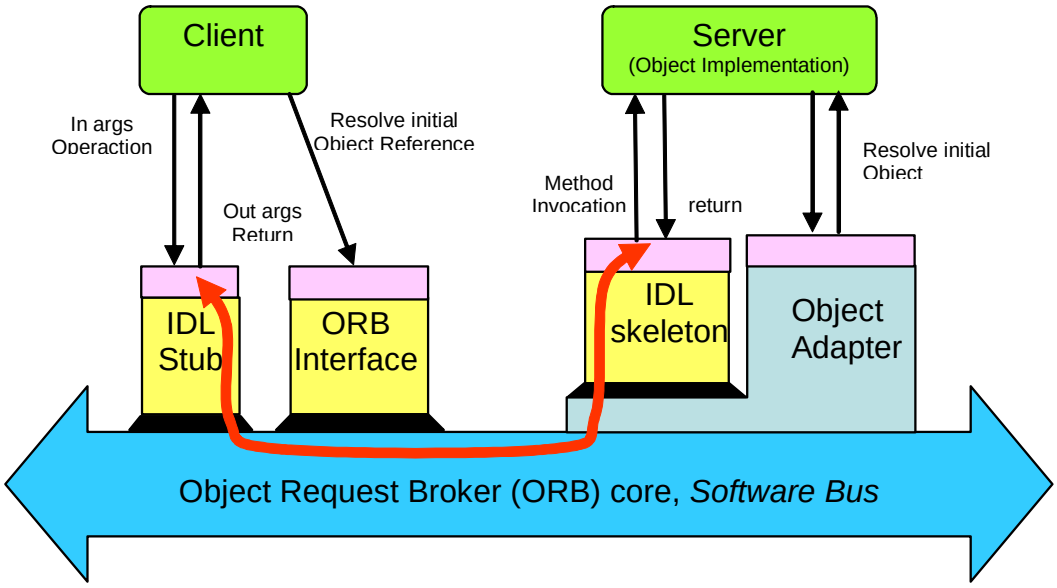


Figure 4.15: Main components and structure of CORBA application diagram.

Communication Features

Although a Request/Reply connection is very common and useful in many situations and applications, there are other communication patterns that can be used to improve the behavior of the the whole system. In Table 4.4 several communication patterns are

Table 4.4: Communication patterns

Pattern	Relationship	Initiative	Service Provider	Communication
SEND	Client/Server	Client	Server	One-way (not blocking)
QUERY	Client/Server	Client	Server	two-way (blocked)
PUSH	Publisher/Suscriber	Server	Server	1-to-n distribution
EVENT	Client/Server	Server	Server	Asynchronous
WIRING	Master/Slave	Master	Slave	Dynamic Component Wiring

described (taken from [Bruyninckx, 2001]). In the table, the Request/Reply pattern is characterized by the QUERY pattern. In all cases (except the WIRING pattern) the service is provided by the Server. The relationship between who initiate the communication and how many are involved in the communication defines the pattern to use.

Selection of a Communication Scheme

Given the above table, we define several variables that can help us to determine which communication scheme is the best for a particular application. The variables defined are:

Number of addressees: This is the number of addressees that will receive the information. Only two options are defined, a) one-to-one, and b) one-to-many.

Type of synchronization: This is a boolean variable and two options are also defined in this variable, a) Synchronous (blocking) and b) Asynchronous (not blocking). In the synchronous mode, the Client waits for the completion of the invocation on the Server side. This is a very risky operation in a distributed environment, so some considerations must be taken into account for de-blocking the process in case a failure come up from the network. Usually, a watchdog timer scheme is used in this case, or an Exception thread can be issued to deal with a big delay. In the asynchronous mode, the client doesn't wait for the completion of the invocation but it must check in the near future if the requested service was completed.

Balance of data transmission (BDT): This is related to the amount of data transmitted in both ways. Defining two levels for this variable (LIGHT, HEAVY), four combinations are possible: a) (LIGHT, LIGHT), b) (LIGHT, HEAVY) c) (HEAVY, LIGHT) and d) (HEAVY,HEAVY). Where the left element corresponds

Table 4.5: Selection of Communication patterns based on number of addressees and synchronization.

Number of Addressees	Synchronization	Communication Pattern
One-to-One	Synchronous (Blocking)	QUERY
	Asynchronous (Not Blocking)	SEND
One-to-Many	Synchronous (Blocking)	Not Available
	Asynchronous (Not Blocking)	PUSH EVENT

to the Client and the right element corresponds to the Server. This variable is used to define a combination of communication patterns.

With these variables we can select one or possible two communication patterns to establish an effective communication behavior. For selecting one communication pattern the following Table 4.5 is used.

For selecting a mixture of communication patterns, we use the BDT variable. This variable is useful only when an unbalance is present on the communication. Then, only a) (LIGHT, HEAVY) and b) (HEAVY, LIGHT) combinations are verified. In option (a) the Client sends a specific request that is answered by bringing big amount of data. This is the common case for searching a topic into the Internet using any searching machine. But the information requested can be answered in a periodic way. In this case, it is much better to have two communication channels, one for requesting and the other for answering. Here the combination of a SEND pattern with a PUSH or EVENT pattern is desirable. The SEND pattern can be used as a START/STOP of the PUSH or EVENT pattern. This configuration is developed in this thesis work for managing the vision system. See appendix C for more details. In option (b) the Client sends a big package of information and only receives as response a short answer. This is the case when many data records are processed to give a true/false answer or to identify a specific object from a database list. The mixture of patterns for this case could be a PUSH or EVENT to send the information to many Servers, instead of many

Clients, then each Server processes the information according to its main functionality, and next a short answer is passed through a SEND pattern. Figure 4.16 depicts this idea used in [Guedea *et al.*, 2002]. In appendix A there is a more detailed description of CORBA services, we can see that PUSH and EVENT patterns are the same as the Publisher/Suscriber defined in this appendix.

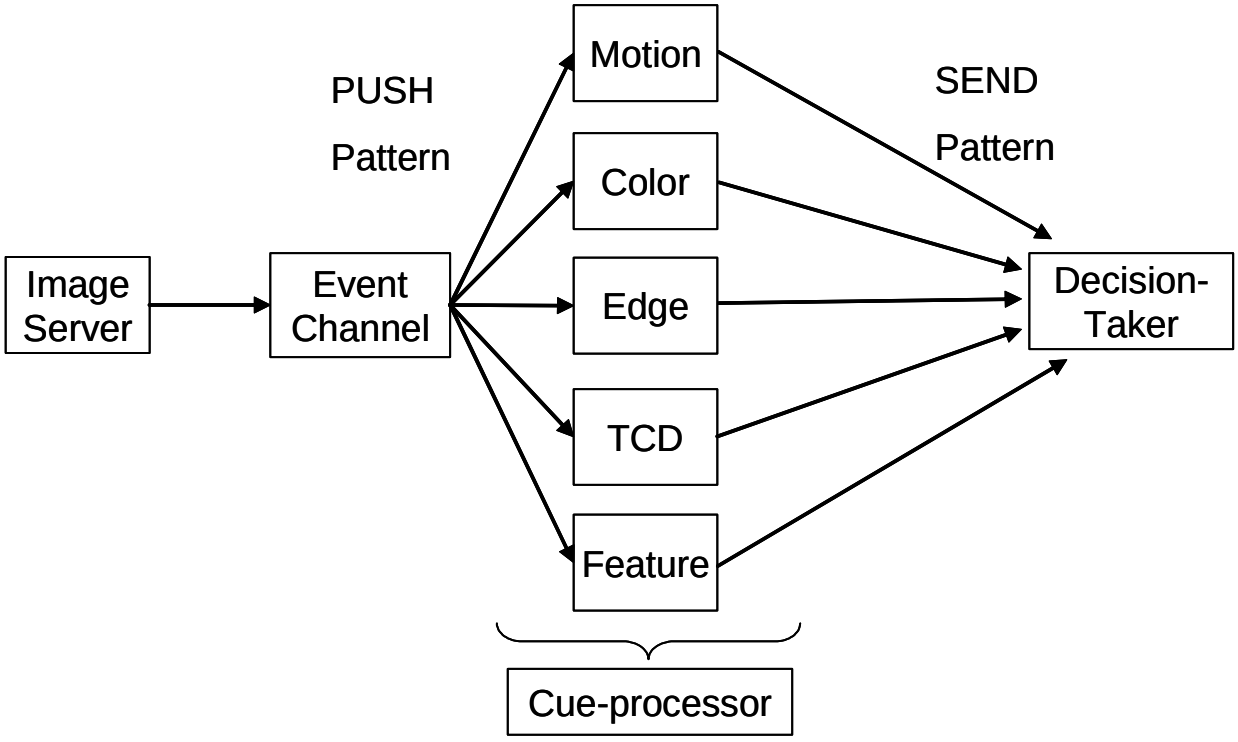


Figure 4.16: Mixture of communication patterns for Cue vision processing.

A final matrix for communication design is created in this stage. The matrix contains the components and their connections in both row and column. On each matrix intersection, the column at the left corresponds to the source component, meanwhile the element at the first row corresponds to the component receptor. Then based on the characteristics of this connection a communication scheme is selected. In our design, we are not considering the WIRING case due to this represents a dynamic connection among components which is out of scope for this dissertation. An example of how this matrix would look is shown in figure 4.18. In this example, three components are defined. The component 1 has two output connections, connection one is with component 2 using the QUERY scheme, and the other connection is with component N using SEND scheme. Component 2 only has one output connection but it can receive messages from component N. Component N, connects with the components 1 and 2 in a *Publisher/Suscriber* scheme (see figure 4.17).

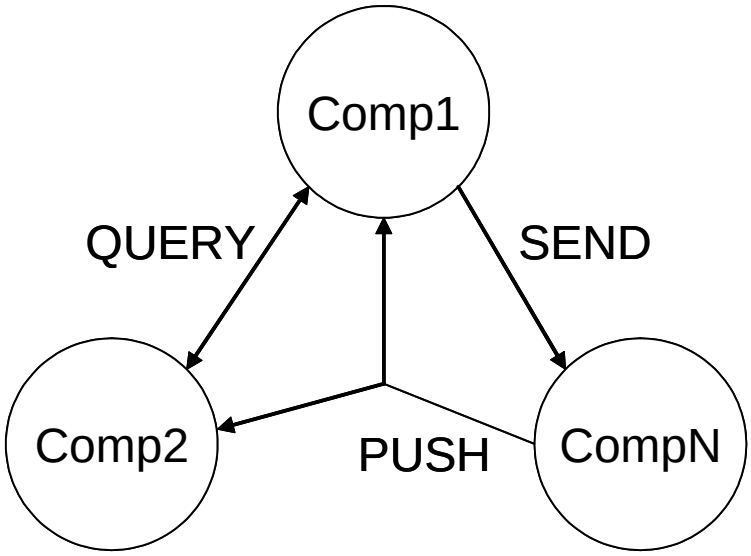


Figure 4.17: Mixture of communication patterns in a small example.

	$Comp1_1$	$Comp1_2$	$Comp2_1$	$Comp2_2$	$CompN_1$
$Comp1_1$	NA	NA	QUERY	NA	NA
$Comp1_2$	NA	NA	NA	NA	SEND
$Comp2_1$	QUERY	NA	NA	NA	NA
$CompN_1$	NA	PUSH	NA	PUSH	NA

Figure 4.18: Communication design matrix

4.4 Wrapper Components Features

The standard features for these components are reusability, connectivity, generality and flexibility. We are adding *abstraction*, and *manipulation*. CORBA specification allows us to achieve connectivity at least for three aspects:

- platform, i.e. actual computational architecture,
- operating systems
- programming languages

Abstraction and reusability are related properties. Reusability looks for making interchangeable modules, while abstraction helps to create these modules. The more abstract a module is, the more general its definition is. On the other hand, if we can manipulate a component *on-the-fly*, then we can achieve certain degree of flexibility. Figure 4.19 shows the conceptual scheme of using Wrapper Components to create application modules. In the example the modules are defined for a mobile robot. The border line between components resembles the concept of a special interface among the components. These components can connect each other using CORBA middleware. The functions of each component are well defined and isolated; we can replace them by a new version or a new brand.

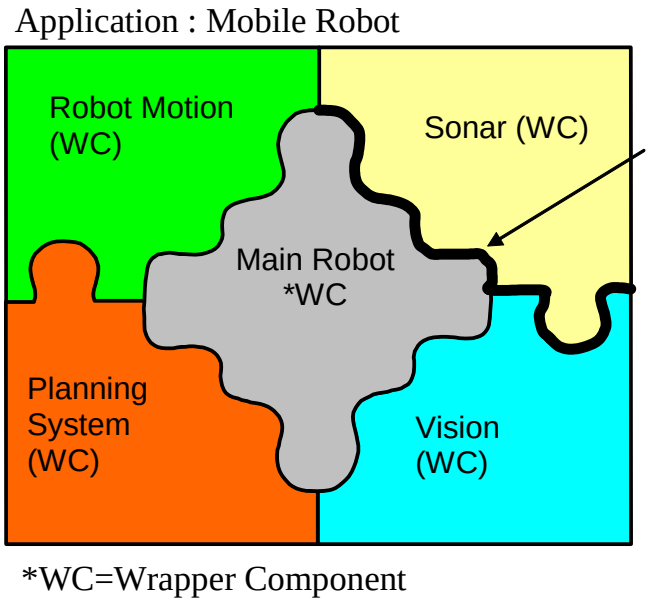


Figure 4.19: Conceptual scheme to create an application using Wrapper Components.

4.5 Summary

In this chapter two general approaches to build wrapper components are described: operation-based and concept-based approaches. For intra-organization development it is better to use operation-based approach. For developing of robotic application we suggest to use concept-based approach. Next, our proposed methodology of three steps: *Divide, Wrap and Connect*, is described as a tool to create wrapper components that will be easier to integrate into the whole system by using the CORBA specification. When we apply this concept to the robotics area, in which several robots could be accessed from different locations and users, or where these robots must realize a collaborative work, it is necessary to have a loose coupling. This leads to the use of concept-oriented components, and more specific, to integration of wrapper components. In the following chapter many of the ideas presented in this chapter are implemented or developed to create a set of wrapper components that will be able to tackle a classical problem from the area of Artificial Intelligence.

Chapter 5

Building an Intelligent Distributed Robotic System

The aim of this thesis is to provide a methodology to create complex robotic applications by integrating distributed and heterogeneous components. If one or several components could perform tasks that belong to the area of Artificial Intelligence, then we can expect that the created system will perform intelligently. In order to demonstrate these ideas, we created a smart distributed robotic system that is able to perform a task in the well-known block-world scenario.

5.1 Problem Definition

A set of blocks is given and each block has a geometric figure centered on one of its sides (see Figure 5.1). None of the figures is repeated. To start the job, the robotic system receives a command from the operator (or user) that is to arrange the blocks in a specific order (final goal). The robotic system must respond if, given an initial state the goal is reachable or not. In case the goal is reachable, the robot arm manipulator proceeds to achieve the new block arrangement (final goal).

5.2 Dividing into Distributed Components

There are many ways to attack the previous problem. Each one depends on the available resources, hardware and software constraints, strategies, and individual or collective knowledge from the personnel involved. We developed and structured the problem according to our resources: a CRS-F3 6-DOF robot arm manipulator, a Videre Design

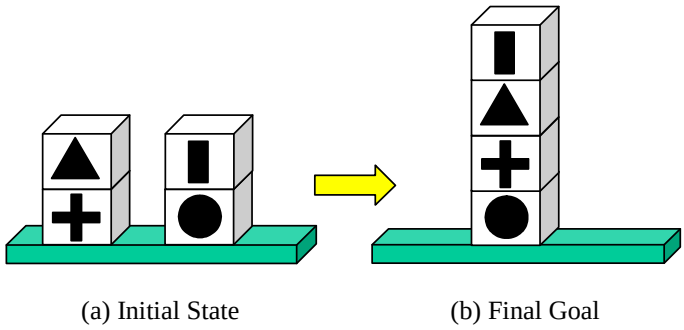


Figure 5.1: Classical block-world problem.

stereo camera, a Pan-Tilt unit where the camera is mounted, a set of generic x86 platform computers, and a mixture of operating systems (Windows 2000, Windows XP, QNX). Some of them are shown in figure 5.2 with the names associated to each computer.

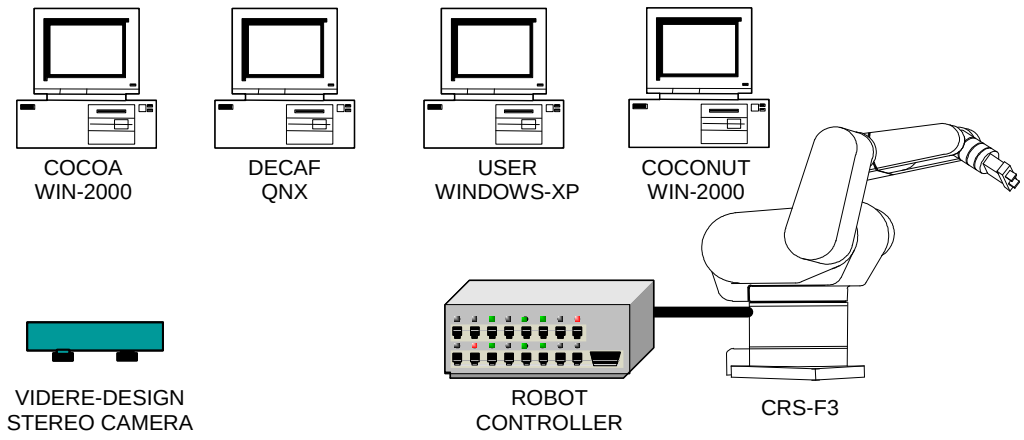


Figure 5.2: Heterogeneous and distributed robotic components.

In order to achieve the final goal (block arrangement) without any human intervention, the distributed robotic system must perform the following tasks: object recognition, planning, and movement of the arm according to vision feedback. It must also react to unexpected changes in the environment. Given these conditions, we started our components definition applying the first step of our proposed methodology: ***Dividing.***

5.2.1 Dividing the Distributed Robotic System into modules

There is a natural or physical separation among some components but there are also others components that are intangible. First, we start with the physical separation of three components: a) The robot arm controller, b) the stereo video camera and c) the Pan-Tilt unit. The arm controller and the Pan-Tilt unit come with their own CPU and

operating system. Both of them provide an Application Program Interface (API) to connect with them through a serial connection. In order to improve the performance and to test the benefits of our approach, we decided to attach two different CPU's and operating systems to each one. The robot arm controller will be accessed through a CPU using a WINDOWS 2000 operating system, meanwhile the Pan-Tilt unit will be accessed through a CPU using a QNX operating system. The video camera is provided with a high-speed communication media but it needs a IEEE 1934 card or chipset into the CPU. In our case, we used a IEEE 1934 card and we also select another CPU besides de CPUs used for robot arm and pan-tilt unit. In this part of the selection process we could select to use the same CPU attached to the robot arm, due to the libraries or API functions were developed for WINDOWS and for UNIX. But after some preliminary tests we found that the demanded resources of the image processing step is high and it could affects the performance of the robot arm. At last we have 3 CPU's attached or assigned to three different hardware resources. This partition is based solely on the physical and performance criteria. For each one of this hardware component, we define three different servers: the Robot server, the Vision server and the Pan-tilt server. A more detailed description is exposed below and there are two appendixes B and C for Robot server and Vision server, respectively.

As we mentioned the previous partition is based on physical and performance criteria, but we need a component to plan the sequence of actions to achieve a specific goal. Then a Planning server is devised, this is a software component and it can be considered as an abstract entity. Furthermore, we found that in order to synchronize all activities, at least one component must provide the function of coordinator. Although, at first glance this partition looks obvious, we had other two options over the table. One option, named option (A), would increase the robot server capabilities by integrating the coordination and planning functionalities into one single component. This will reduce the number of components but also it will reduce the chance of exchange components. Figure 5.3 depicts this idea.

The other option (B) is less integrated and only consider the coordination capability included with the motion control of the robot. The planning function is considered as another component. This scheme reduces the centralization aspect of option (A) and increases the number of interconnections among the components. There are four components at level-1 and two subcomponents at level-2. Figure 5.4 depicts this idea.

Finally, we also have to establish that the components must interact among themselves using a *Client-Server* approach. Until now, we have only defined the servers side, but it is necessary to include one or several clients that start the operation of the overall system. One basic client is the component that allow the operator or user to control and

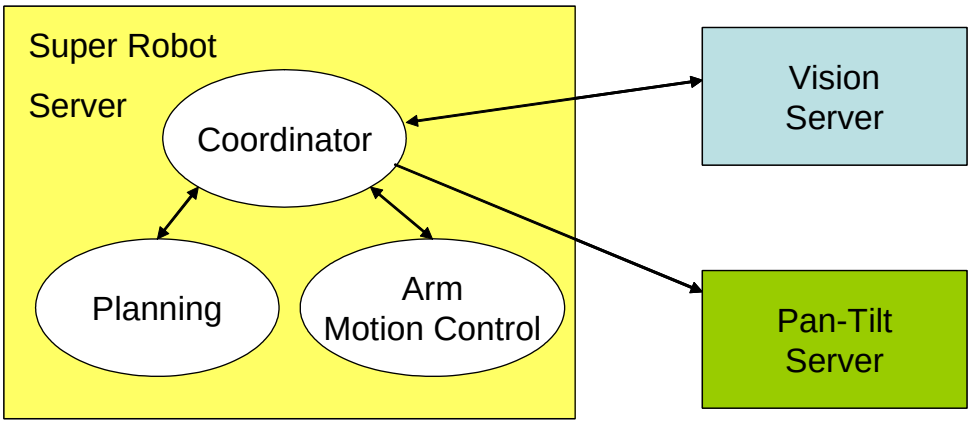


Figure 5.3: First step of proposed methodology, Option A. In this case a Super Robot server is defined. The communication links are reduced and at the same time more centralized. three components of first level are defined and three subcomponents at level-2. Client modules (components) are not included

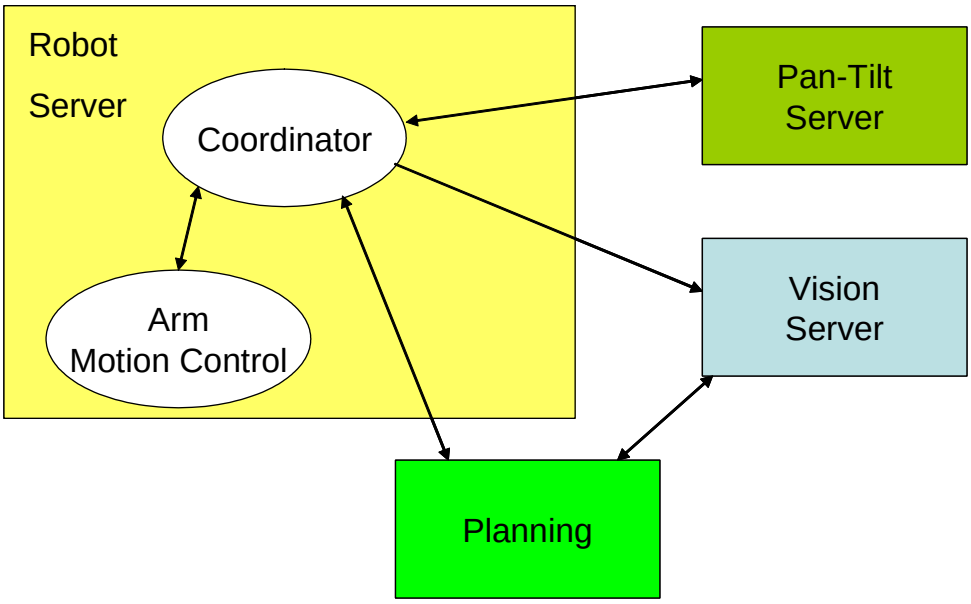


Figure 5.4: First step of proposed methodology, Option B. In this case a more divided approach than option A is considered. The communication links are more distributed.

to configure the operation of the system; we referred it as the User Interface or Graphical User Interface (GUI). Next, we have some implicit Client-Server relationships between some components. For instance, the coordinator is a client of the planning server and of the vision server. The planning server is also a client of the vision server, as we will shown later. The final result, is a set of servers and clients which have a specific function. Figure 5.5 shows the final division with the main components that we defined for this application, Pan-tilt server is not shown in this figure. Here, we decided to separate the coordinator component from the arm motion controller. This separation will give us the option to change the robot server depending on the robot to manage. Furthermore, we will have the opportunity to change the coordinator according to the problem to attack without worrying about which robot is attached.

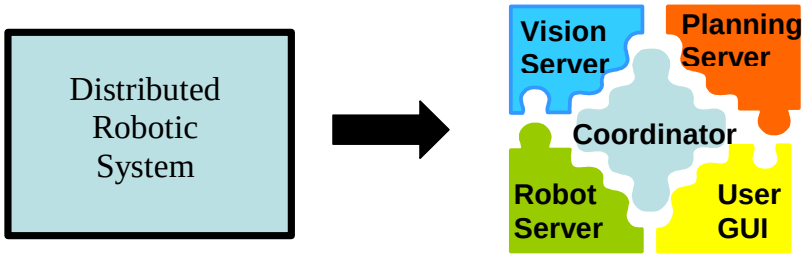


Figure 5.5: First step of proposed methodology. Final division, in this case several clients and servers are defined at level-1. Different form options A and B, here the coordinator is managed as another component

Following we will describe briefly what are the main services and aspects of each component. Next section gives more details on the interface for each component, and appendices B, C and D give more detailed information for Robot Server, Vision Server and Planning Server, respectively.

Robot Server.

This component hides all intrinsic details to move the arm manipulator and provides a generic interface to manipulate any kind of arm manipulator. In order to achieve this flexibility, an abstract interface for any kind of arm manipulator is defined on the next section. This approach makes high level design easier but increases the programming complexity for this server. Although the robot server acts as a "zombie" (just follow commands without analyzing), it has some real-time issues that makes its programming a challenge. A more detailed explanation is described on Appendix B.

Vision Server.

This component provides four basic vision tasks: Image Acquisition, Image delivery (or transmission), Object Recognition and Object Tracking. Each task is managed as a different wrapping level as we will explain in section 5.3.2. These tasks can be requested by other components such as, the coordinator, the planning server and/or the User GUI. The amount of data and the frequency of each data set is managed using different communication schemes, this is explained on Chapter 6. A more detailed explanation about how this services are implemented is provided in Appendix C.

Planning Server.

The main function of this server is to provide a sequence of actions in order to achieve a given final goal from an initial state or condition. This sequence of actions is possible, if and only if the final goal is reachable. In order to generate a plan, the planning server must request some information from the vision server in order to ascertain the initial condition. Next, a sequence of actions is delivered to the coordinator component. The main challenge in the development of this component is the translation between different data formats and the synchronization with the other components.

Coordinator.

In other articles we named this component as a "brain controller" [Guedea *et al.*, 2006b] or "task controller" [Guedea *et al.*, 2004], but due to the different activities that it realizes, we renamed it as the "coordinator" of the robotic system. Basically, its main functions are to start and control the robot movements according to the actions sent by the planning server and the visual information received from the vision system. Previously, the coordinator passed along the information about the final goal to the planning server. This goal is received from the GUI component.

User GUI.

This component is an artifact to provide an interface between the robotic system and the user or operator. It aids in configuring each one of the components of the robotic system, so instead of receiving information from a single source, the client makes connections with the different servers in different ways, as we will explain later in the next chapter.

As we mentioned, there is a physical separation among some components but because of the location transparency provided by CORBA specification, some components can be working in the same computer (to save resources). We can also have the components separated (to increase concurrency). In our case we tried to further separate the components in order to improve the distributed issue. It shows how CORBA can manage this situation seamlessly. The final arrangement for these components is shown in figure 5.6, and the outcomes for this stage are shown in Table 5.1.

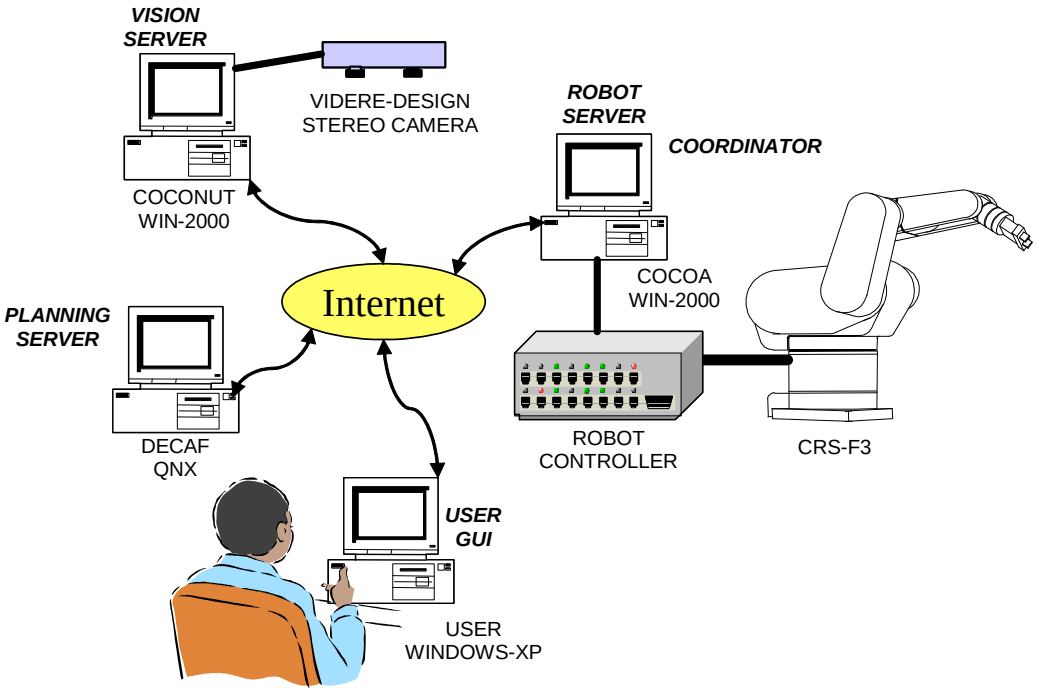


Figure 5.6: Final arrangement for our Distributed Robotic System.

Resulting Indexes

Given the previous definitions for our distributed robotic system the following indexes are computed according to Section 4.3.1:

Complexity Index

$$I_{cpx} = \frac{\sum N_{wc_1}}{\sum N_{wc_1} + \sum N_{wc_2}} = \frac{6}{6+0} = 1 \tag{5.1}$$

This number indicates a low interchange of components among them for this project, but any component can be changed without affecting the definition of the others.

Structural Index

Table 5.1: Outcomes of the dividing process for the Distributed Robotic System.

Component Name	**Component level (1,2)**	**Physical Constraint**	**Performance**	**Abstract Functionality**	**Connections**
Robot Server	Level-1	√	NA	NA	1
Vision Server	Level-1	√	√	NA	3
Planning Server	Level-1	NA	NA	√	2
Pan-Tilt Server	Level-1	√	√	√	1
Coordinator Client	Level-1	NA	√	√	4
GUI Client	Level-1	NA	NA	√	3
Total Number of Components	Total Components for each level	Total Physical Components	Total Performance Components	Total Abstraction Components	Total Number of connections
$N_{wc} = 6$	$N_{wc_1} = 6$ $N_{wc_2} = 0$	$N_{pc} = 3$	$N_p = 3$	$N_f = 4$	$N_c = 14$

$$I_{struct} = \frac{N_{pc} + N_p + N_f}{N_{wc}} = \frac{3+3+4}{6} = 1.667 \tag{5.2}$$

This number reveals that some components are divided using more than one criteria.

Connection Index

$$I_{cnx} = \frac{\sum N_{c_i}}{N_{wc} * (N_{wc} - 1)} = \frac{14}{6 * 5} = 0.467 \tag{5.3}$$

This number indicates that some of the components have more than one single connections, creating a moderated mesh of connections.

5.3 Encapsulating the components

In the methodology proposed the second step is to *encapsulate (wrap)* each component, so they can interact with each other through a standard middleware specification, in this case CORBA. The main results of this step are the IDL interface of each component and this implies also the communication scheme to use. The wrapping process must deliver an interface with the following properties: *Abstraction, Monitoring and Configuration.* We will describe how these properties are developed in the Robot Server, the Vision Server and the Planning Server. Coordinator and User GUI can be seen as the clients of the servers and they do not provide a specific service to wrap. Pan-tilt server is explained in next chapter due its simplicity, and because it is used only on the first

stage of the development. This unit is kept fixed for the following stages in order to facilitate some computations.

5.3.1 Robot Server Interface

For this thesis an arm manipulator is used as the robot component. Our goal is to define an interface that would be abstract enough, so that we can use the same interface for most arm manipulators types. In general, we can visualize any arm manipulator as

a) a set of links and junctions (Operational approach) and

b) a copy of the human arm (Conceptual approach)

In the first case, the interface could be abstract enough to deal with most kind of arm manipulators but the high level design effort increases too much. In the second case, we can command the arm as if we are playing the "put the tail on the donkey" game, i.e. we can extend, retract, move the arm left or right, just to mention basic movements. An example of the above statements is shown in Figure 5.7.

```
Interface robot {
     Void Move_Joint(int id, int joint, int dist);
     int get_dim(int id, int joint);
}
```

(a) Operational Approach

```
Interface robot {
     Void Extend(int dist);
     Void Retract(int dist)
     Void Move_Left(int dist);
     Void Move_Right(int dist);
}
```

(b) Conceptual Approach

Figure 5.7: Operational approach and Conceptual approach when defining an interface for a robot server.

From the previous example, the implications in the operational approach is that client must know which axis to move depending on the displacement desired. Meanwhile in the conceptual approach there is only a distance to move. Furthermore, moving axis by axis it may be possible to reach a specific point but it requires a big effort to make a straight move just using the operational approach. On the other hand, using Move_left() or Move_Right() the robot can move several axes at the same time, but this is hidden to the client.

Other difference in the implications could be the type of robot (cylindrical, polar, spherical, SCARA or gantry), the number of axes , must industrial spherical robots has 6 axes and SCARA robots have 4 axes, but they can have an extra axis for a track or

one axis less, see Figure 5.8. The number of axes defines which axis will be the wrist where the end-effector will be installed. Because of all this implications, we defined the following function set for a generic arm manipulator: (see Figure 5.9). In this interface definition, functions *Speed()* and *Learn()* are used to configure some variables into the robot, meanwhile *Monitor()* and *Finish()* are two methods used to monitoring the status of the robot. Function *Speed()* configures the global speed of the arm manipulator and function *Learn()* is used to create a set of previously defined locations reached by the arm. Function *Monitor()* returns the current real-time position of the robot in an axis-based format, and function *Finish()* reports if the robot has finished its last movement.

With this interface definition we have 17 functions or methods. For the Robot CRS-F3 which has 182 functions in total, the Abstract Index using only these numbers is:

Abstraction Index

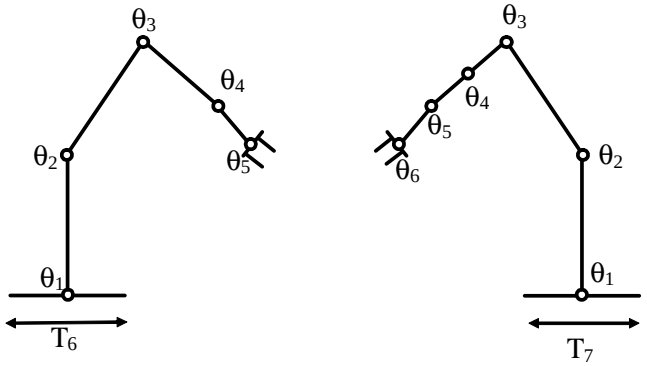
$$\text{CRS-F3 } I_{abstract} = \frac{17}{182} = 0.093 \tag{5.4}$$

this represents a big effort on the codification of the robot server. For the Motoman Robot this number is

Abstraction Index

$$\text{Motoman } I_{abstract} = \frac{17}{82} = 0.207 \tag{5.5}$$

this represents a less effort than the CRS-F3.



(a) T265 (b) F3

Figure 5.8: Two spherical robot from CRS Robotics with a different number of axes and different position of wrist axis.

```
// IDL Robot Definition
interface Robot {
// Basic function to get the status of the robot
void get_status(out string status);
// Basic function to command an action to the robot
void do_action(in string action);
////////////////////////////////////////////////////////////
// COMMANDS TO MOVE THE ARM MANIPULATOR
//

void Retract(in long distance, out boolean result );
void Extend(in short direction, in long distance, out boolean result );
void Turn(in short direction,in short degrees, out boolean result);
void MoveH(in short direction, in long distance, out boolean result );
void MoveV(in short direction, in long distance, out boolean result );
void Turn_EF(in short direction,in short degrees, out boolean result );
void Turn_Wrist(in short direction,in short degrees, out boolean result );
void Home(out boolean result );
void Ready(out boolean result );
void Speed(in short velocity, out boolean result );
void Learn(in long var, out boolean result );
void Goto(in long var, out boolean result );
void Gripper(in long dist, out boolean result);
void Monitor(out float  x, out float  y, out float  z,
             out float rx, out float ry, out float rz,
             out float j1, out float j2, out float j3, out float j4,
             out float j5, out float j6, out float j7, out float j8);
void Finish(in short option);
};
```

Figure 5.9: Basic Robot IDL interface definition to command a generic arm manipulator.

5.3.2 Vision Server Interface

Encapsulating computational vision services is a though issue. There are different stages during the image processing and each one provides a specific service for others components.

Computational vision can be divided as three sequential main tasks: (see Figure 5.10)

- Image Capture and Enhancing,
- Feature Extraction or Segmentation and
- Data interpretation

Each task can be "wrapped" and due to they are sequentially executed, the later one encapsulates the previous one. This provides different wrapping levels. A low level corresponds to the first task, Image Capture and Enhancing, and a higher level corresponds to Data interpretation. Usually low levels correspond to large data processing meanwhile high levels correspond to small data processing but more complex computation.

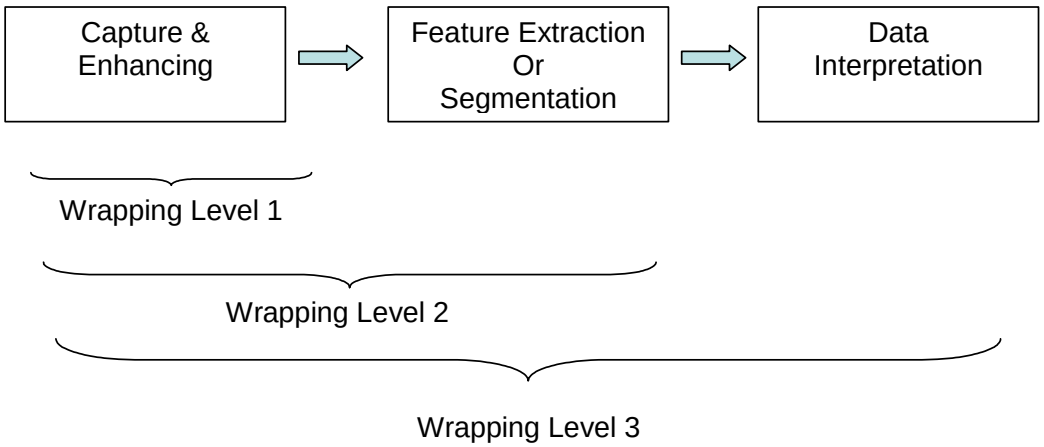


Figure 5.10: Different wrapping levels for computational vision tasks.

Each task may have its own interface, and according to the equipment used (physical camera) it may also have its own setup parameter list. But all these configuration levels increase the complexity of the vision server. So, instead of thinking in how many ways an equipment could be adjusted, we considered the main services we can have from any vision system (abstract functionality). Furthermore, the services requested must be aligned with the required application. In this particular application (block-world problem) there is a need to recognize a set of objects and their location based on pixel information.

In this research, the vision system will be used with other robotic components to enhance the dexterity of an arm manipulator. Given this scenario, it is observed that the response time of the vision system must be in the range of seconds (less than 2 seconds) or milliseconds, instead of tens of seconds or minutes. It is necessary to reduce the computational effort of the algorithms to match the required speed for specific tasks. In order to meet this requirement, some limitations are imposed into the environment and the objects around the arm manipulators. These limitations come from the specific hardware currently installed in the laboratory and some imposed constraints (structured environment).

The vision system besides providing a fast response, it also has to realize different task in order to be useful for other distributed components. These tasks are classified as

- Object Recognition, and
- Object Tracking

They correspond to the higher wrapping level 3, i.e. data interpretation. We decided that each task can be executed locally (through a console scheme) or remotely. The last option must be done using a high-speed network connection to meet the response time requirement.

In order to recognize a set of objects, the vision system must learn the main characteristics of these objects, and it is difficult to do this without supervision. For this reason there is a special command (*learn()*) directed to the vision system indicating which object must be learned. Once the object is learned and its main features are stored in an object database, it can be located just asking for it by name. In this case, the feedback result is true or false depending on whether the object is found in the current image stream or not. If the result is true then its current position is returned.

Finding an object is an expensive task in computational terms. The vision system has to look for all similar objects and then it makes a careful selection after checking more parameters. This procedure is full described in appendix C. So, in order to save time and to meet the time requirements we provide another function called *Track()*. In this case the premise is to look for the object in the last reported position. This search is faster and different from *find()* command. Tracking an object is a function that works temporally and it can be started several times for the same object or different objects. The vision system keeps a list of all object requested and sends the information in a periodic and structured way through a specific channel. This is done using one of the services provided by CORBA, event service. This communication is stopped using the command *EndTrack()*. Finally, due to the system could learn many kinds of objects, there is a special command to inform which are the learned objects. This command is

named *Get()*.

Given this context, we used a wrapping level 3 to define the vision server commands and to hide the internal details about how the system store, retrieve and process the image information. This IDL interface is shown in Figure 5.11. Furthermore, the commands are defined in a non-block manner. This avoid possible conflicts when several objects are requested at the same time by several clients. In the next chapter we will describe how to deal with the integration issues.

```
// IDL Vision Server Definition
interface  ImageServer
{
            void Learn(in short x, in short y, in string name);
            void Find(in string name);
            void Track(in string name);
            void EndTrack();
            void GetObj();
};
```

Figure 5.11: Basic Vision IDL interface definition to command a generic vision server

5.3.3 Planner Server Interface

Planning is one of the activities that exposes a certain degree of intelligence, due to the selection of actions under specific conditions and goals [Russell and Norving, 2002]. Research in this area has been done for manufacturing plants in order to optimize the use of resources and timing constraints. There are many types of planning problems and approaches. Examples of planning problems are configuration planning solving for packing pallets into a truck, route planning in networks, path planning of robot, and solving puzzles. Task planning has different challenges when applied to robotics. An intelligent robotic unit has to decide the next action given the current situation of the environment. The context information is acquired by mean of robot's sensory system and/or through a specific command. Although the robot or the set of robots have a main goal to achieve, there are specific situations that need to be managed first, possibly in the opposite way to the final goal. For example, let us consider a case where there are several robots trying to "hunt" a prey in an environment with many obstacles. Suppose the robots know the exact position of the prey. Even under these simple considerations, the robots need to make a path to avoid the obstacles between them and the prey, and they also need to avoid paths where they block themselves. A global planner for these situations can be useless because of the stochastic behavior of the environment. In this case a reactive planner or multi-path planner must be necessary to improve the behavior

exposed by the robotic units.

In this work, we developed a generic planner using the Graphplan developed by Blum et.al.,[Blum and Furst, 1997]. Graphplan is a planner based on STRIPS-like domains which uses a compact structure called Planning Graph. This planner always returns the shortest-possible partial order plan, if it exists, or states that no valid plan exists. To generate desirable actions to accomplish given goals, we need to represent a given problem. A fact describes a particular situation. An action, operator, describes how the action changes the given facts before and after. A goal is a set of facts that should be true.

The ***Blocks-World*** is a classical example of artificial intelligence research. It consists of labelled blocks; in our experiments (see Chapter 6.0), different shapes are attached on blocks, such as a circular shape and a triangle shape. In the formulation of this problem, there is a manipulator that can perform the following actions: *pick up, put down, stack, and un-stack*. Unlike simulated experiments in artificial intelligence, in our experiment we actually use a robot manipulator that performs these actions with the help of visual feedback from a stereo camera. Figure 5.1 shows a picture of the block world with shape−labelled blocks. As we described before, the robot server does not have the previous commands defined in this section. For this reason we called these actions as Macro-tasks, and they are executed using micro-tasks. These micro-tasks are actually the commands defined on the robot server interface and vision server interface, and they are requested by the coordinator module. Next chapter will explain in detail this approach.

In the STRIPS-like planning domain, operators have pre-conditions, add-effects, and delete-effects, all of which are conjuncts of propositions, and have parameters that can be instantiated to objects in the world. Operators do not create or destroy objects, and time is represented by events. In a planning problem we have:

- A STRIPS-like domain (a set of operators)
- A set of objects
- A set of propositions (literals) called Initial Conditions
- A Set of Problem Goals which are propositions that are required to be true at the end of a plan.

An action, is a fully-instantiated operator. For instance, the operator 'Grasp ?x' may instantiate to the specific action 'Grasp block-A'. An action taken at event t adds to the world all the prepositions which are among its *Add-Effects* and deletes all the propositions which are among its *Delete-Effects*. For example, *pick-up (x)* operator has precon-

dition of *on(x, Table)*, *clear(x)*, and *arm-empty*. The effect of *pick-up(x)* is *holding(x)*. Figure 5.12 shows this operator in STRIPS language.

```
(operator
 PICK-UP
 (params (<ob1> OBJECT))
 (preconds
            (clear <ob1>) (on-table <ob1>) (arm-empty))
 (effects
            (holding <ob1>)))
```

Figure 5.12: PICK-UP operator description in STRIPS language.

There are different types of planning systems such as Partial-Order-Planner (POP), Graphplan, and SATplan. POP uses least commitment search space. Graphplan exploits relaxed problem, then searches. SATplan translates to logic, and then uses satisfiability algorithms. In our research, we adapt Graphplan, because it always returns a shortest possible partial-order plan [Blum and Furst, 1997]. Some of the advantages of Graphplan are that it always finishes and it is faster than POP. The main idea of Graphplan is to solve a relaxed problem. It converts the information to propositional representation. Then it constructs a graph with levels which have time steps. Graphplan, then, identifies simple inconsistencies between pairs of actions.

Design and Integration of Cooperative Planning System

Since our main research focus is to integrate multiple robots and to build cooperative robots, we have decided to use an existing graphplan program. In order to use existing code and to reduce the complication of using the code, we designed a wrapper planner component. The wrapper component hides all the details of the planning system, and provides outside interface to other robots and modules.

As shown in Figure 5.13, to wrap the existing planning system program i.e. graphplan algorithm code, we need to have two components. The first component is a data conversion component. In this case, it converts visual information in LISP format from the vision server (see details in appendix C) and from the user (through the coordinator module) into data that the planning system can understand. The second component is data communication component, with this the outside module can ask for planning operation and retrieve the actions that accomplish the planning goal. Through this interface the clients send their information (facts, operators and objects) and request the plan to achieve a specific goal. In appendix D we expand the explanation of this IDL interface which is by far more elaborated than previous interfaces. Figure 5.14 shows the complete IDL definition.

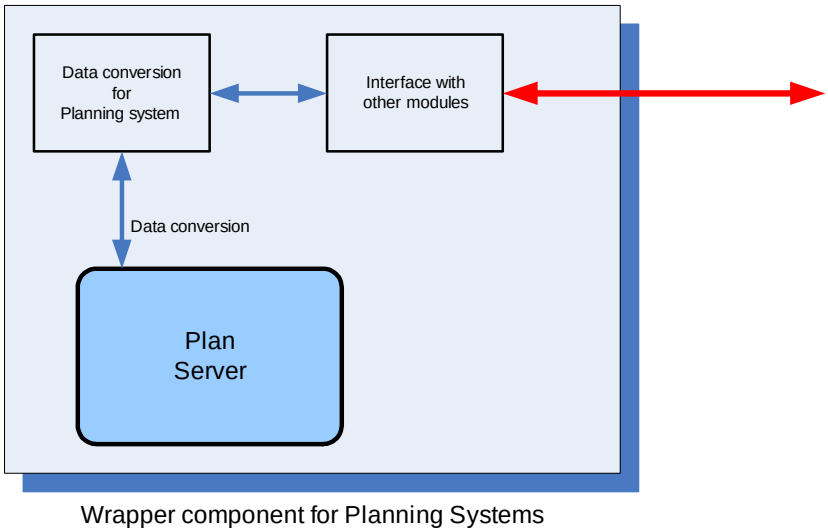


Figure 5.13: Wrapper component for planning system

5.4 Connecting the components

Once each component is defined and wrapped, the next question is how to connect them. In this stage we will apply the third step of our methodology: *Connecting*. Connecting components can be seen as assembling a puzzle (Figure 5.15) or assembling a lego toy. In the first case the components are assembled following certain pattern matching in color and shape, meanwhile in the second case the assembling is realized following a given path or sequential operations. We will use the second case, and we will provide an incremental approach in this integration. By using the *client-server* approach under CORBA this is a straight way to connect them and it is already implemented. But depending on the application we can use other communication schemes, as they are described on section 4.3.3.

Our final goal is to tackle the *block-word* problem, and in order to achieve this goal we split the problem into smaller subproblems. Basically, we devise three subsystems:

- a remote-operated robot,
- a remote-operated vision system, and
- the final autonomous robotic system

for each one, there are some specific details on the services provided by the servers that other scheme rather than client-server approach is required. Following are the communication schemes selected for each subsystem.

```
module GPLAN{
    struct Token   {    string item;    };
    typedef sequence<Token> tokenList;

    struct Fact    {    tokenList item; };
    typedef sequence<Fact> factList;
    typedef sequence<Fact> paramList;
    typedef sequence<Fact> precondList;
    typedef sequence<Fact> effectList;
    struct Operation
    {
            string name;
            factList param;
            factList precond;
            factList effect;
    };
    typedef sequence<Operation> opSeq;
    struct Facts
    {
            factList types;
            factList initials;
            factList goals;
    };

    struct Plans    {   string name;    };
    typedef sequence<Plans> planList;

    interface OpInterface {
            // read operation file and save into OP_LIST
            void getOperations (inout opSeq OP_LIST_clt );
            // read facts file and save into FACT_LIST
            void getFacts (inout Facts FACT_LIST_clt );
            // read facts file and save into PLAN_LIST
            void getPlan (inout planList PLAN_LIST_2clt);
            void startPlan ();
    };
};
```

Figure 5.14: Planning IDL interface definition

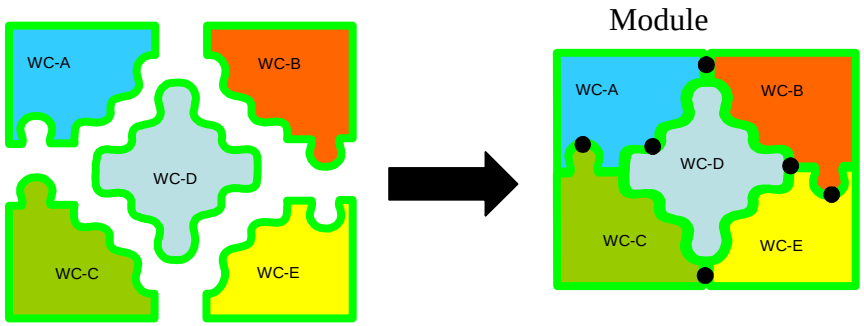


Figure 5.15: Third step of our methodlogy: **Connecting** components

5.4.1 Connecting a remote operated robot

In this subsystem we visualize three main servers and one single user, see Figure 5.16. The robot server is waiting for commands to start its movements. The communication between the robot server and the user follows the *client-server* approach. The user starts the communication by sending a specific command to the robot server, then depending on the command it could wait for an answer. Same case is for the communication between the pan-tilt server and the user, although in this case is one-way. The relationship between these two components is more of the MASTER-SLAVE type. The user assumes that the pan-tilt server executes the command. Meanwhile, the communication between the vision server and the user is managed as a *publisher-subscriber* approach. In this scheme we use the Event service provided by CORBA, see more detailed information in Appendix A. We selected this type of communication because the information transmitted through this channel is the image frame captured at specific rate. This represents the wrapping level 1 for the vision system (see Section 5.3.2) where only capturing and enhancing processes are realized.

The resulting communication matrix for this subsystem is shown in Table 5.2. The GUI component has three connections, but one connection is not directly established, meanwhile the other components only have one connection. The communication between the user and the vision server can be managed as a PUSH or EVENT type. Last option represents a client-server communication based on events, but eventually the vision server can attend or send the image frame data to any other component connected through the channel. In this way it is better to define it as a PUSH type communication. Taking this issue into consideration, it can be visualized that there is not explicit interface between the user and the vision server. Instead of this, there is a set of specific interfaces to connect to an Event channel. These interfaces are already provided in CORBA specification. The only information needed is the IOR address of the Event Channel.

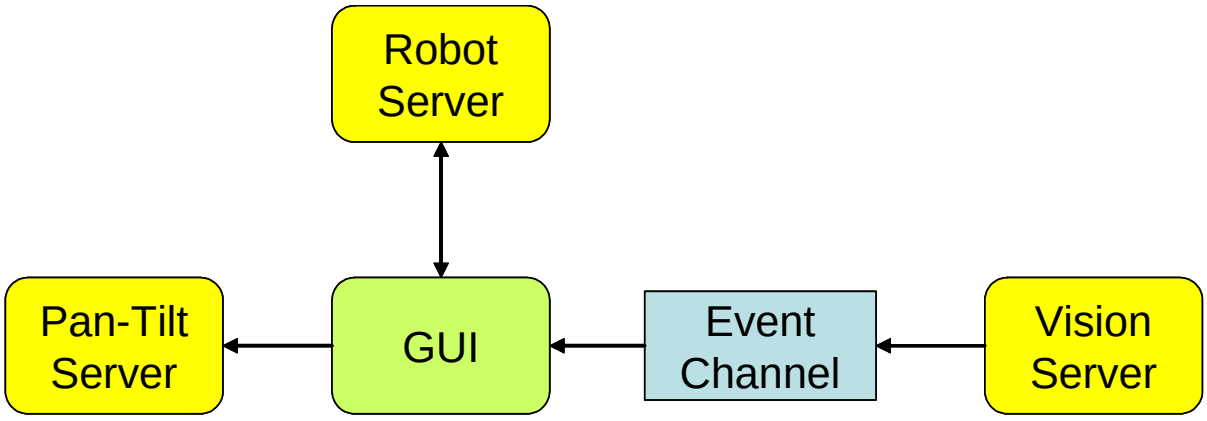


Figure 5.16: Communication scheme for a remote operated robot.

Table 5.2: Communication design matrix for a remote operated robot

	GUI	**Robot Server**	**Vision Server**	**Pan-Tilt Server**
GUI$_1$	NA	QUERY	NA	NA
GUI$_2$	NA	NA	NA	SEND
Robot Server$_1$	SEND	NA	NA	NA
Vision Server$_1$	PUSH	NA	NA	NA
Pan-Tilt Server$_1$	NA	NA	NA	NA

5.4.2 Connecting a remote operated vision system

In the previous subsystem, there is not a explicit need to manipulate the vision server from a remote location, although a basic configuration change could be pursued such as the frame rate of the capturing process. Even though, there are some services defined in section 5.3.2 that can be requested from a remote location. In this case the client could be the user (operator) or other component defined previously, such as the *coordinator*. We distinguish two types of communication, which are related to the type of information transmitted and the frequency and operation of them. Basically, there are image data and object data information transmitted. First one, image data transmission, is started from the vision server console and it uses a specific channel to send this information to all possible clients connected through this channel. Second one, object data transmission, is started under request of a client. This request is always of a SEND type and the information is transmitted through a second event channel. Depending on the command, the information transmitted could be the answer for a simple question (QUERY) or the START/STOP action of a sequence of periodic information (PUSH). The final communication scheme for this subsystem is shown in Figure 5.17 The resulting communication matrix for this subsystem is shown in Table 5.3.

Table 5.3: Communication design matrix for a remote operated vision system

	GUI	**Vision Server**
GUI$_1$	NA	SEND
Vision Server$_1$	PUSH	NA
Vision Server$_2$	PUSH	NA

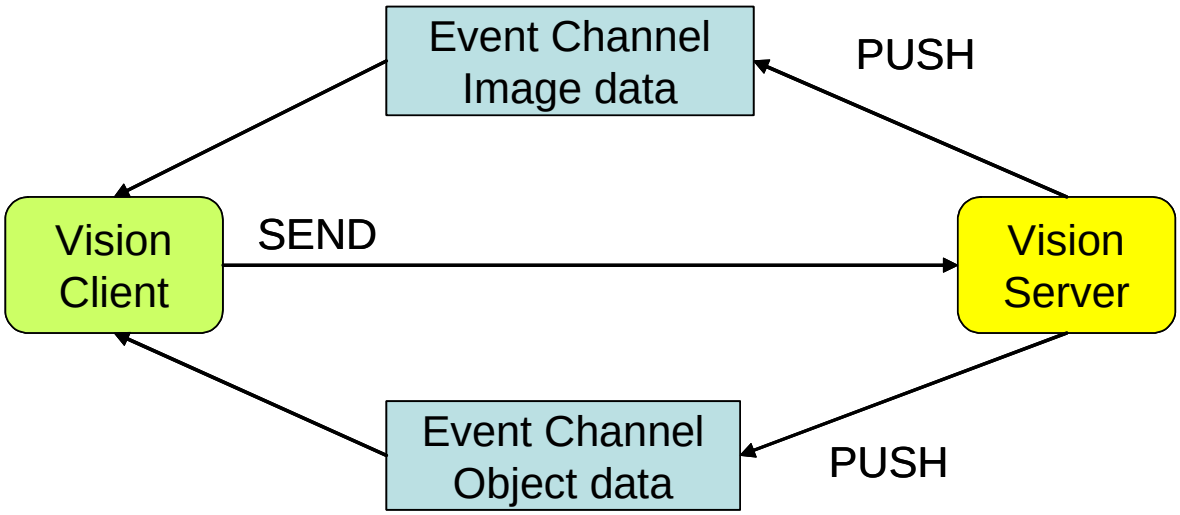


Figure 5.17: Communication scheme for a remote operated vision system.

5.4.3 Connecting an autonomous robotic system

Based on the previous subsystems, two more components are integrated with the two previous subsystems in order to create the final robotic system. These components are: the planning server and the coordinator client. The coordinator realizes the orchestration of the different components and provides also an user interface to manipulate the vision system. The planning server is included to plan the sequence of actions given an initial state and a predefined set of operators. The initial state is given by the coordinator to the planning server. Then, several new interconnections are devised on this new arrangement. In Figure 5.18 it is shown in detail the information transmitted on the new arrangement. First at all, the coordinator replaces the functions realized by the user in the two previous subsystems. Next, the new connections are established with the planning server. Basically, there is a connection with the planning server to SEND commands with information for the planning server or a request for starting the planning process. In this final arrangement the coordinator has 6 connections, but two of them are implicit. This refer to the connections established through the event channels.

5.5 Summary

Through this chapter we described how a complex robotic system can be divided into different modules, and how they are transformed into wrapper components to facilitate the design process and to provide inter-operability among them using different communication patterns. The IDL interfaces for robot server, vision server and planning server were defined. Also, two different subsystems were analyzed separately and next they were integrated into a final robotic system. The main concepts for each component

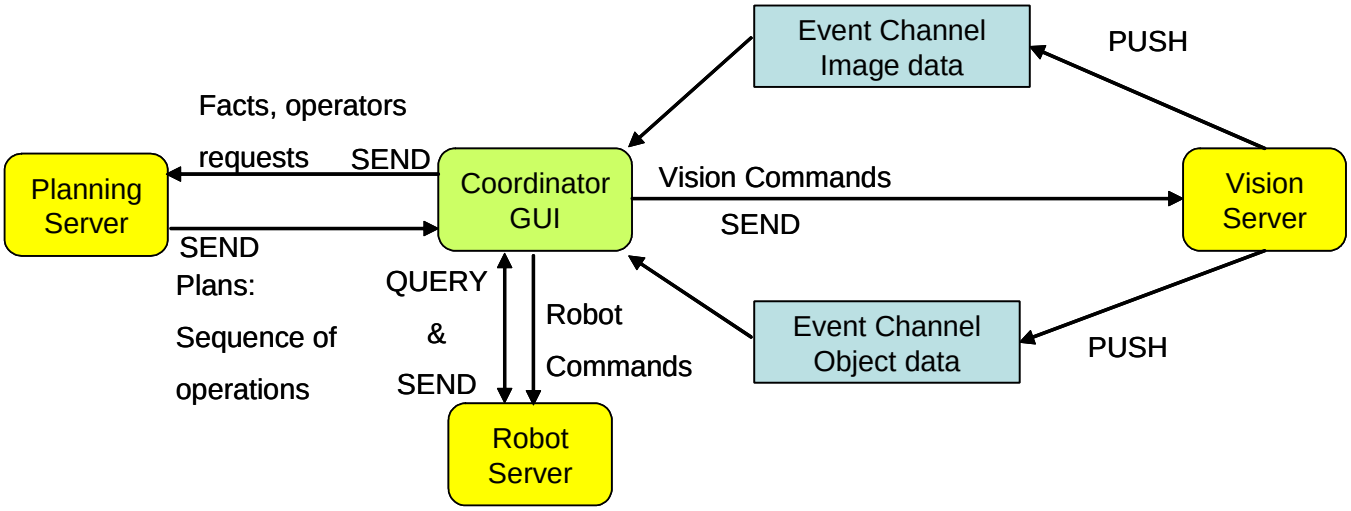


Figure 5.18: Communication scheme for the autonomous robot system.

Table 5.4: Communication design matrix for an autonomous robot system

	Coordinator GUI	**Robot Server**	**Vision Server**	**Planning Server**
Coordinator GUI$_1$	NA	NA	SEND	NA
Coordinator GUI$_2$	NA	SEND	NA	NA
Coordinator GUI$_3$	NA	QUERY	NA	NA
Coordinator GUI$_4$	NA	NA	NA	SEND
Vision Server$_1$	PUSH	NA	NA	NA
Vision Server$_2$	PUSH	NA	NA	NA
Robot Server$_1$	QUERY	NA	NA	NA
Planning Server$_1$	SEND	NA	NA	NA

and stage were exposed and more specific details can be founded in the appendices B, C and D. Even though, the integration and testing of these components represents a big effort if we decided to integrate them in a single step. To overcome this challenge, next chapter deals with this task using an incremental approach which will be explained through a set of experiments.

.

Chapter 6

Experimental Setup and Results

In previous chapter we disassembled the problem of the block-world case into smaller subsystems following the three basic steps of our methodology: *Dividing, Wrapping and Connecting*. These subsystems were developed independently but with a common objective. This approach reduces the complexity problem. The idea is to proof that modular components with limited capabilities can be assembled together with less effort if they share some basic standards mechanisms. At the end, with simple functions the overall system could behave in a better way. Now, we will describe in some detail the experiments carried out to test each subsystem first and then the final integration. The subsystems can be developed using an approach of *Console-Server*. This means, that each module can be checked and tested without using any communication scheme at the beginning. Indeed, this saves a lot of time during the debugging process due to it is not necessary to establish a connection with the client side.

The experiments corresponds to the following subsystems:

1. Create a remote operated robot
2. Create a remote operated vision system for object recognition and tracking
3. Create an autonomous robot system

6.1 Integration of a remote operated robot

In this experiment the goal is to move a robot through the internet. This means, the user is located in a remote place. It is not at the same location of the robot. In order to move any kind of robot (SCARA, Polar or spherical) a generic user interface is developed. This interface has the commands defined for the robot in previous Chapter

5 (see Figure 5.9. Also, a vision system with its pan-tilt unit is required. The unique function of this system is to provide a visual feedback of the robot movements. We want to test the connectivity of the system (image transmission and command transmission) and the manipulation of the remote robot. The way how the robot is executing every command is explained in Appendix B.

6.1.1 Description of main components

This work was presented at [Guedea *et al.*, 2003a]. In this case, we integrated a robot server, an image server (just for image transmission) and a pan-tilt unit (see Figure 6.1). Table 6.1 shows the physical or mechanical limits of the 6 DOF robot (CRS-F3) and the number of axis. There is a track of 5 meters where the robot displaces. This track has assigned the axis number 7. The main issue is to manipulate the robot arm remotely using CORBA as the middleware bus. The user interface concentrates and interacts in one window with three components: the robot arm, the pan-tilt unit and the image server.

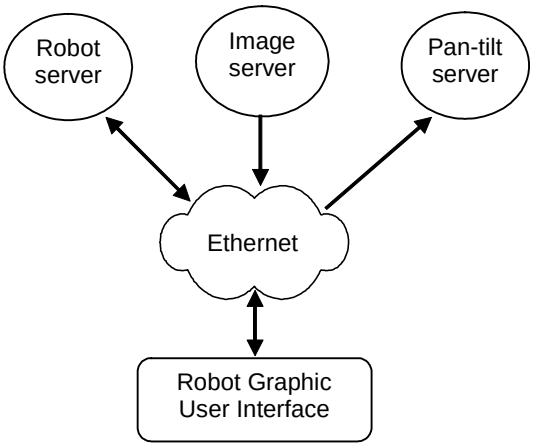


Figure 6.1: Main components of a remote operated robot arm.

The interaction between the robot server and the user interface is of the type client-server. Most of the methods supplied by the robot server are manipulated using spring slides into the user's window design or single buttons for single actions. Only the slide representing and configuring the robot speed is static. Figure 6.2 exposes these features.

In Figure 6.3 there is an image of the robot server console. Here the user can start and stop the robot server. Each command received on the robot server is displayed and the current position of each axis is shown. The button "Get Control" means that the Server will take control of the robot controller instead of the teach pendant. The button "Start Server" starts the Robot Server and after this step all commands can be accessed

Table 6.1: Physical limits and number of axes for the robot arm manipulator

Axis	**CRS-F3**
1	-180° to +180°
2	-135° to + 45°
3	-135° to +135°
4	-180° to +180°
5	-135° to +135°
6	51 turns or ± 18,432 °
7	5000 mm

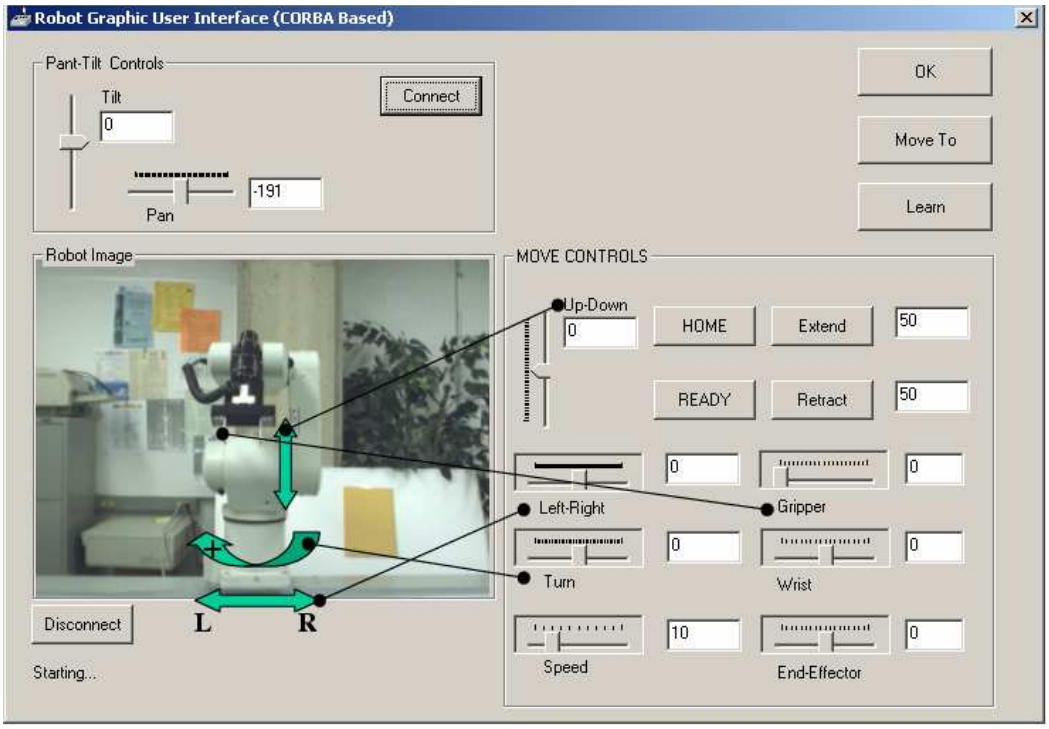


Figure 6.2: User interface used for interaction with a robot arm manipulator.

through the Internet. With this step the `Robot.ref` file is created.

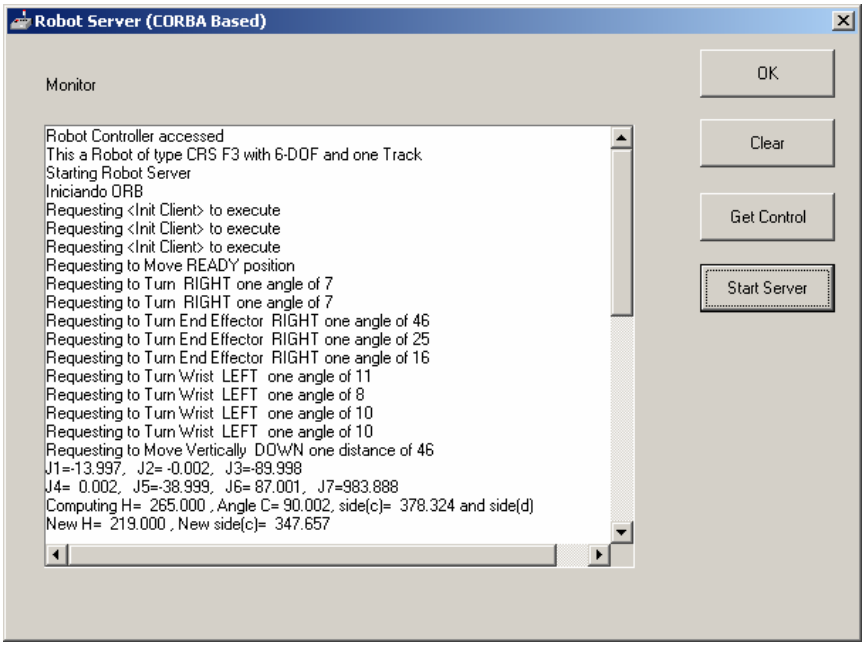


Figure 6.3: Robot Server Console for a CRS-F3 arm manipulator.

Regarding the pan-tilt server, it has few operations and its IDL interface hides all internal conversions. For this experiment we worked with two similar pan-tilt units which have different counting ranges and resolutions, Table 6.2 shows these differences. Figure 6.4 shows the IDL interface. It can be visualized that all commands are of the type SEND, none of the functions returns a value and the same case happen for the parameters of each function.

Table 6.2: Differences on pan-tilt units abstracted into the IDL interface

	PTU-46-17.5	**PTU-46-70**	**Any**
Pan	-3090 to +3090	-12404 to 12403	-159 ° to +159 °
Tilt	-911 to +3090	-3674 to 12448	-47 ° to + 31°
Speed	300°/sec	60°/sec	0-100 % V max
Resolution	3.086 arc min	0.771 arc min	Depends on windows interface

Finally, the image stream is acquired using one service of CORBA specification: Event Service. In this case the image server is always running and sending the image stream through a channel created specifically for this information. It wraps the information according to level 1 shown in Section 5.3.2. The raw image is compressed to JPEG format and the ratio compression is about 10 percent of original size. A schematic image with the ORB as a software bus is shown in Figure 6.5.

```
// IDL Pan-Tilt unit Definition
interface PT_control
{
  void pt_init();           // initialization
  void pt_close();    // close port
  void set_pan_speed(in short speed);
  void set_tilt_speed(in short speed);
  void set_pan_pos(in short pos);
  void set_tilt_pos(in short pos);
  void set_pan_rel_pos(in short rpos);   // relative position
  void set_tilt_rel_pos(in short rpos);  // relative position
  void reset_all();
  void reset_pan();
  void reset_tilt();
};
```

Figure 6.4: Basic Pan-Tilt unit IDL interface definition.

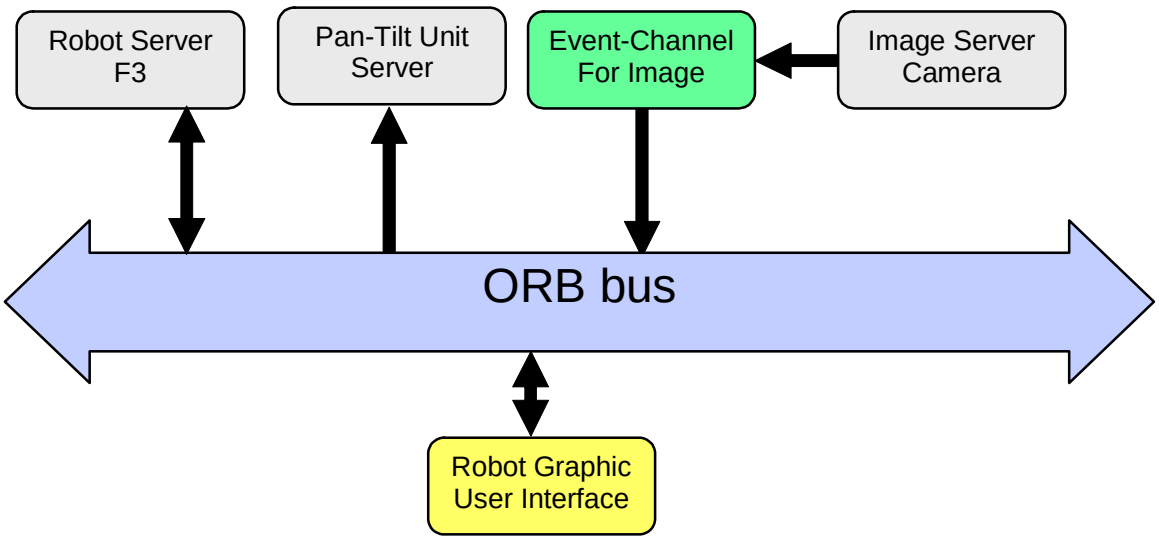


Figure 6.5: Wrapped components needed to build a remote operated robot arm using CORBA specification.

6.1.2 Off-line setup or tasks

This experiment was tested twice; one test was performed with all components executing in a local area network of the Pattern Analysis Machine Learning Lab (PAMI-LAB) (see Figure 6.6) and the other test was realized with the Graphic User Interface (GUI) located in a computer of the Center of Intelligent System at the ITESM, Campus Monterrey (see Figure 6.7). In the last case the distance between the GUI and the servers is approximately 3,000 km. The camera is located about 2 meters distance from the center point of the robot base. In order to capture several images, a slow movement on the robot is configured (speed at 10 % of the maximum range).

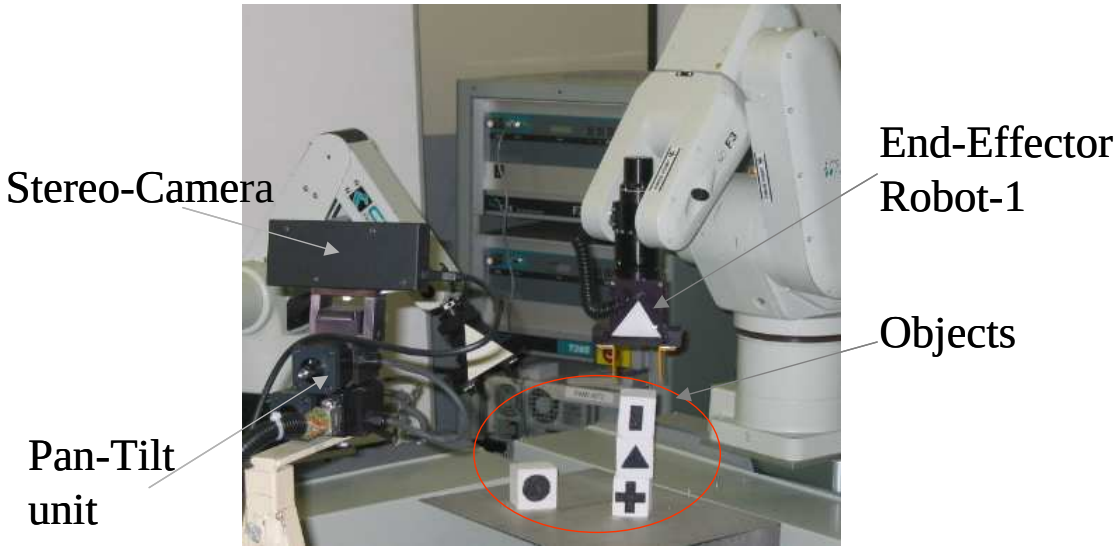


Figure 6.6: Experimental set up at the Pattern Analysis and Machine Learning Lab.

In order to record the information and actions followed by the user (ITESM Monterrey location), a screen capture program (HyperCam) was used in its computer to save a video of the operations. Next, this video movie is processed using a free video editor, see Figure 6.8. The idea is to determine when the frames change on the stream. The capturing is realized using a 30 frames per second rate. The screen capture program is configured under Windows-XP/2000 as a real-time task priority.

In Table 6.3 there is a description of the computational resources used for each component. Vision Server and Event Service were running in the same computer to avoid delays on the transmission. Robot Server and GUI were running on the same computer when the testing was local. Pan-Tilt server was running alone in the QNX machine.

We tested our system using two "flavors" of CORBA implementations. Computers with

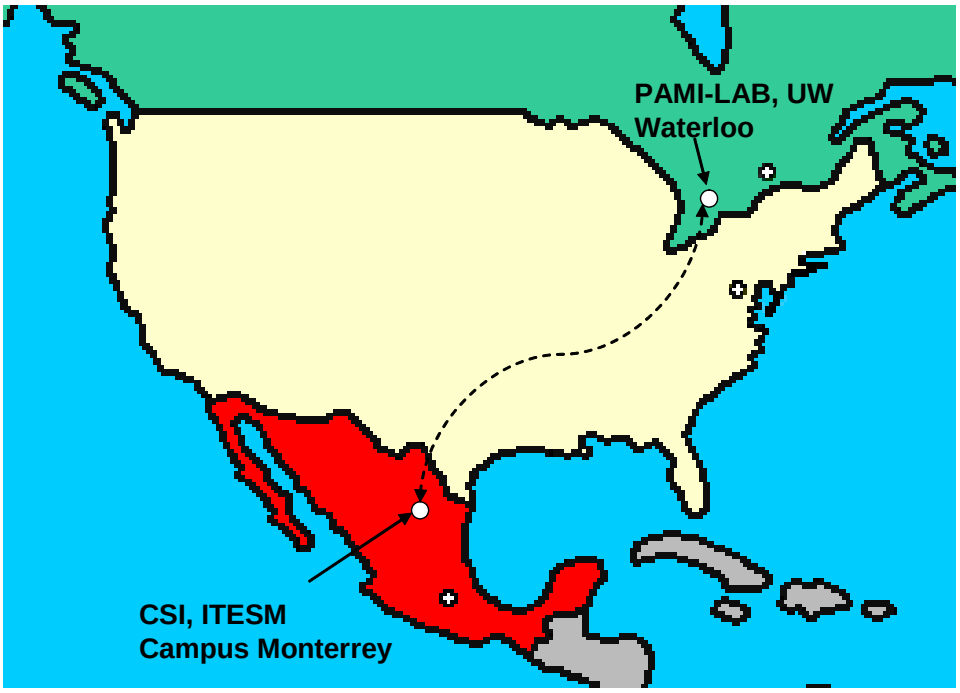


Figure 6.7: Largest distance used for testing the remote operation of the robot. The Vision Server and Robot server are located at the PAMI-LAB, while the Graphic User Interface is (GUI) located at the Center for Intelligent Systems (CSI). The GUI window is captured using a screen capture program (HyperCam).

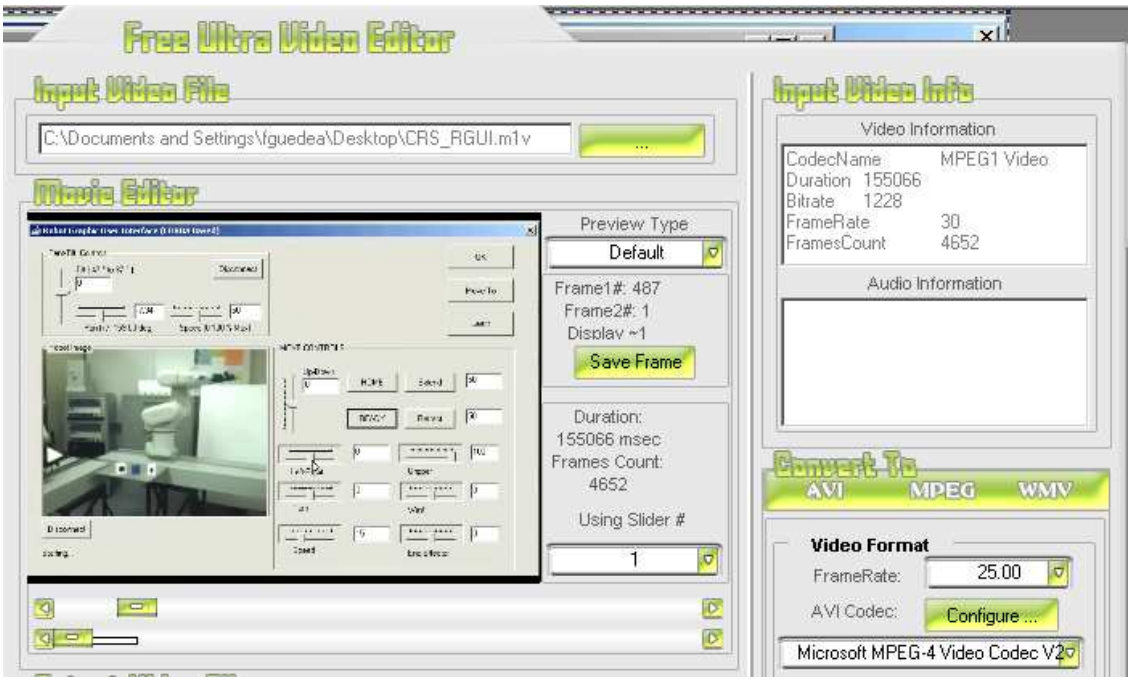


Figure 6.8: Movie Editor for analyzing robot sequence frames

Table 6.3: Computational resources used in a remote robot operation. Most of the computers names are related to coffee flavors.

Component	**Computer Name**	**Operating System**	**Platform**
Robot Server	COCOA	WIN-2000	Pentium-IV 2 GHz, 528 MB
Vision Server	COCONUT	WIN-2000	Pentium-IV 2 GHz, 528 MB
GUI	any	WIN-XP/2000	Pentium-IV 2 GHz, 528 MB
Pan-Tilt Server	DECAF	QNX Neutrino 6.2	Pentium-Pro 200 MHz, 64 MB
Event Service	COCONUT	WIN-2000	Pentium-IV 2 GHz, 528 MB

WINDOWS 2000 have installed the libraries provided by ORBACUS version 4.1, which is compliant with CORBA 2.0 specification. The computer with QNX used TAO version 1.5 (The ACE for ORB) with ACE (Adaptive Communication Environment) version 5.5, which also is compliant with CORBA 2.0 specification.

6.1.3 Starting sequence of servers

The sequence to operate this subsystem is the following:

1. Starting of the Event Channel (IOR for Event is created, `Event.ref`).
2. Starting of the Image Server (IOR file is not required).
3. Starting of the Pan-tilt server (IOR for pan-tilt is created, `Pantilt.ref`).
4. Starting of the Robot server (IOR for Robot is created, `Robot.ref`).
5. Sending all IOR reference files to the user's PC.
6. Starting of the GUI client.

When the GUI client starts it will exit if the robot server (`Robot.ref`) is not available. With the pan-tilt server and Event service it will only check their status but it will not exit if they are not available. In order to connect the GUI client with the servers, it is necessary to send (transmit) all the *.ref files from these servers to the current directory where the GUI client is running. Figure 6.9 depicts this sequence.

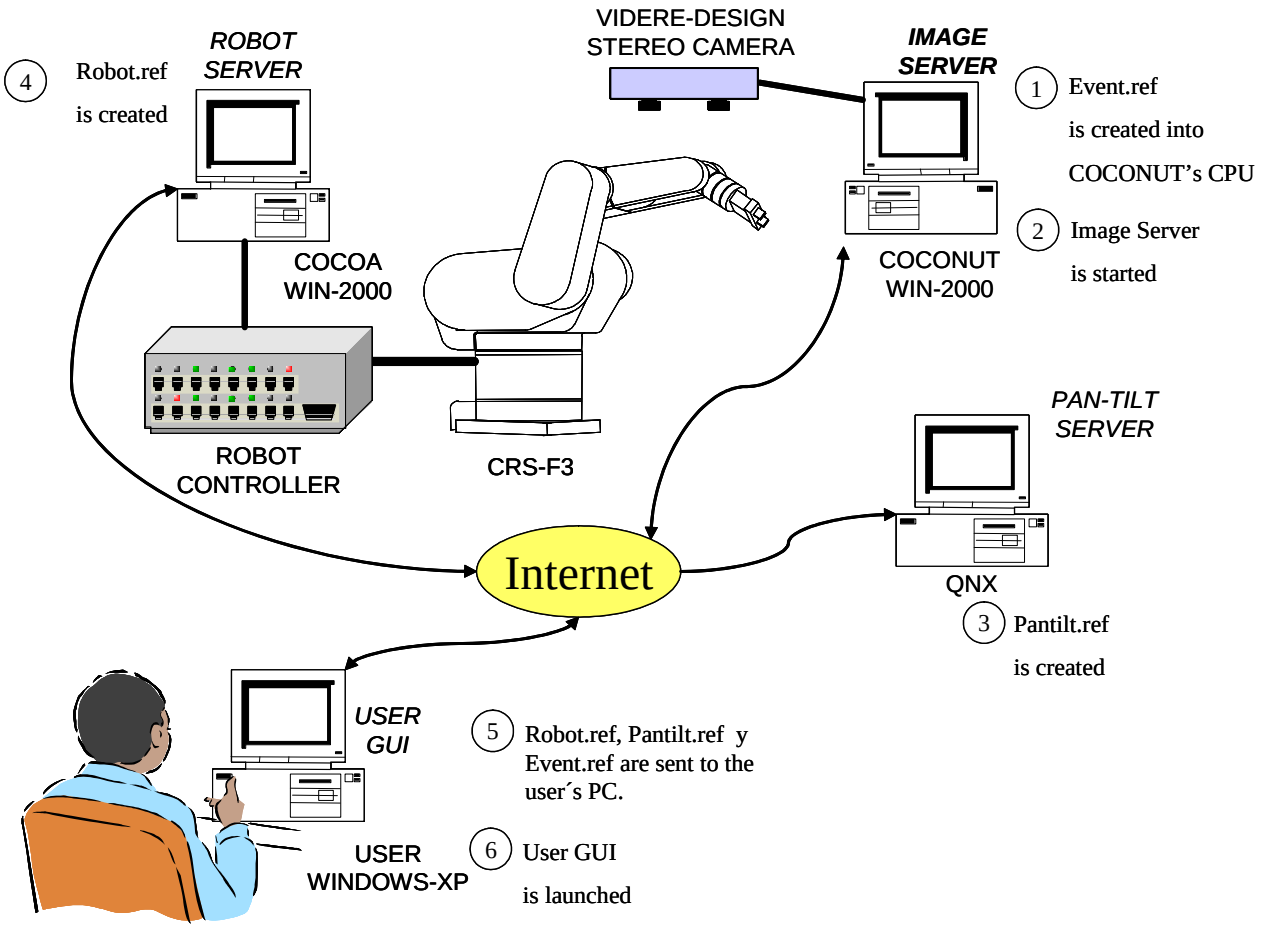


Figure 6.9: Starting sequence for a Remote Operated Robot. IOR reference files are sent to the user's PC before the Client interface is launched.

6.1.4 Main operations

In this subsystem, the operator moves the robot arm remotely using a set of basic commands. These commands can be applied to several robot arm configurations such as: spherical, SCARA and gantry. The operator moves the arm to pick up a piece using the image feedback provided by the vision server. He also can manipulate the pan-tilt unit to move the camera.

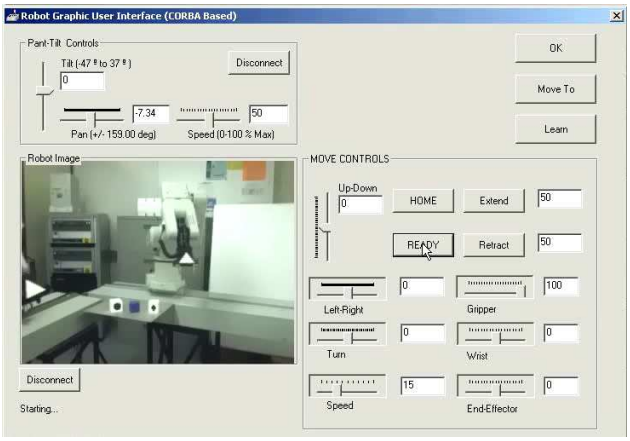
The main issue about CORBA implementation in this subsystem is the management of the Event Service channel and the IOR reference information. The image server is managed through a console and it can connect and disconnect at any time from the Event Channel. It does not care about how many clients are connected. Once the references files are sent to the location of the GUI Client, we started the remote operation of the robot.

Saving current position and moving to ready position.
The first operation is to save the current robot position into an internal variable. Next, the READY button is pressed to move the robot to the ready position. Depending on the robot, this position could be an alignment of the axis to form a vertical alignment or a 90 degree alignment. CRS-F3 robot makes a 90 degree position. The following images (Figures 6.10 and 6.11) shown the information displayed by the movie editor at different frame snap shots. For each change on the image the number frame is registered. The difference between consecutive changes is recorded and the elapsed time is computed. Table 6.4 shows the elapsed time recorded from frame 272 to frame 423. It can be observed that the delay time for the client side is in average 0.56 seconds while there is a maximum lag time of 1 second. There is also a delay time since the robot controller receives the command and the robot starts its movement.

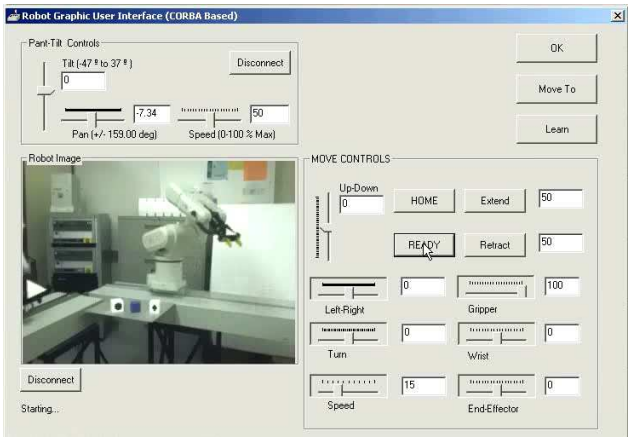
(a)Frame 144 (b)Frame 272

(c)Frame 288 (d)Frame 304

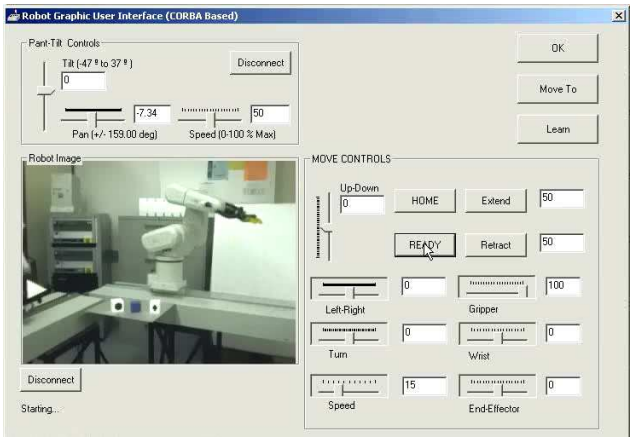
Figure 6.10: SAVE and READY command execution. Part-I. The first two images shown the frame number where the LEARN(SAVE) and READY button are depressed, respectively. The next two frames shown the frames where the robot images in the picture changed.



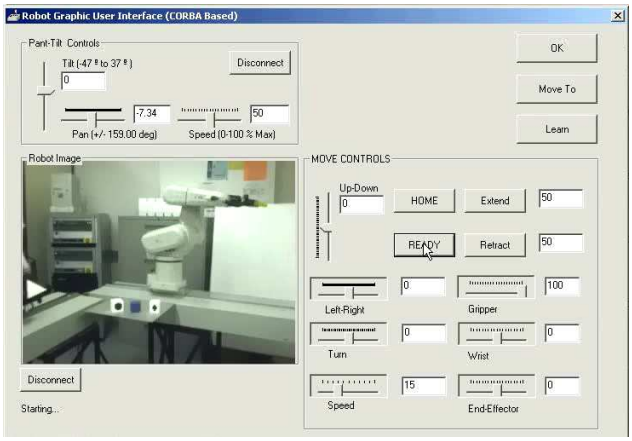
(a)Frame 315



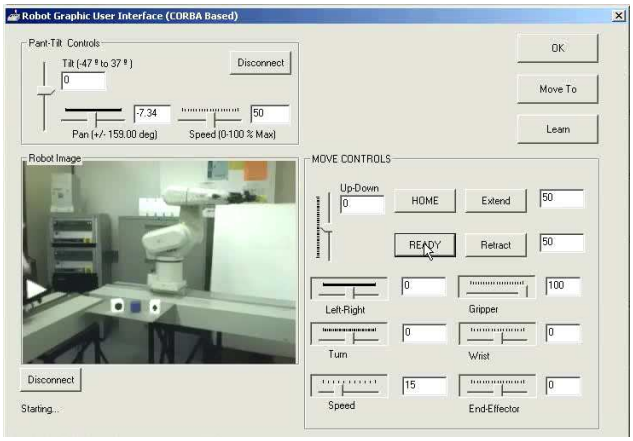
(b) Frame 327



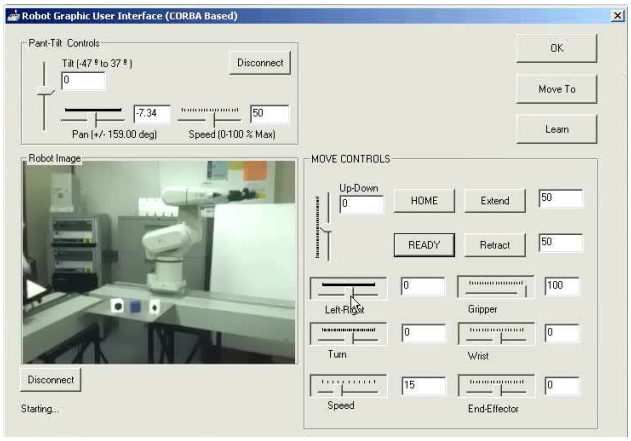
(c)Frame 357



(d) Frame 378



(e)Frame 393



(f) Frame 408

Figure 6.11: SAVE and READY command execution. Part-II. The images shown in the different frames describing when the robot images changes in the small square.

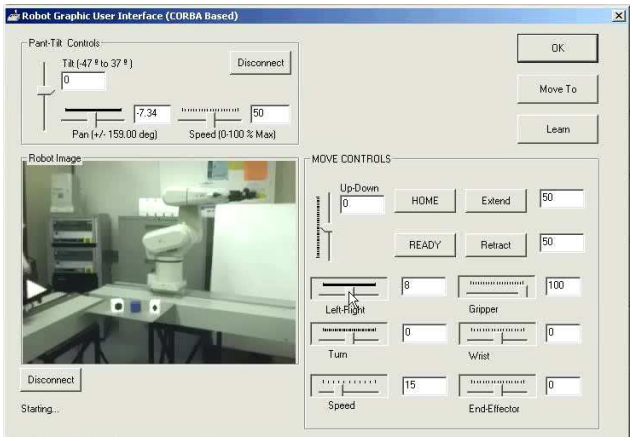
Table 6.4: Frame Timing for video stream of SAVE(LEARN) and READY commands.

Image	**Difference**	**Elapsed time (sec)**	**Event description**
Frame 144	-	-	LEARN button depressed
Frame 272	-	-	READY button depressed
Frame 288	16	0.533	1st change of robot position
Frame 304	16	0.533	2nd change of robot position
Frame 315	11	0.367	3rd change of robot position
Frame 327	12	0.400	4th change of robot position
Frame 357	30	1.000	5th change of robot position
Frame 378	21	0.700	6th change of robot position
Frame 393	15	0.500	7th change of robot position
Frame 408	15	0.500	8th change of robot position
Frame 423	15	0.500	9th change of robot position

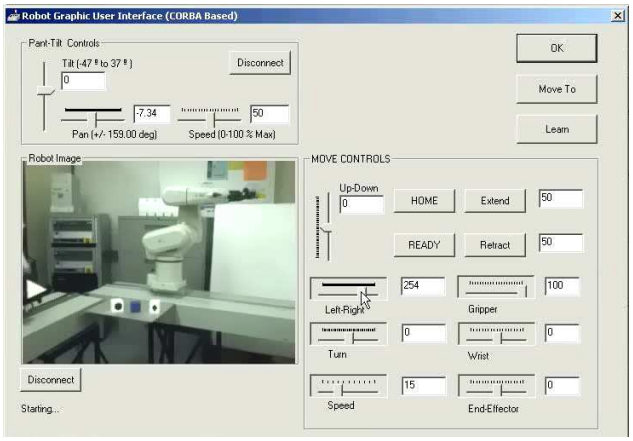
Moving robot with different commands

Next operation is to move the robot using different commands such as: Move Horizontal (left or right), Move Vertical (Up or Down), Extend/Retract the arm. The following images (Figures 6.12 and 6.13) shown these movements. In figure 6.12, Frame 514 shows the instant when the slide is pressed. Next, the slide is moved to right until the number 254 is shown on the text box (Frame 587), this means 254 millimeters. Frame 589 shows when the slide is released, at this moment the value at the display returns to zero. The following frames shown when the robot starts the movement. The number of frame indicates when the image is updated on the interface. In figure 6.13, it can be visualized how the end-effector is getting out of the view. Next section shows the frame shots where the pan-tilt unit is moved to keep the robot inside the image view.

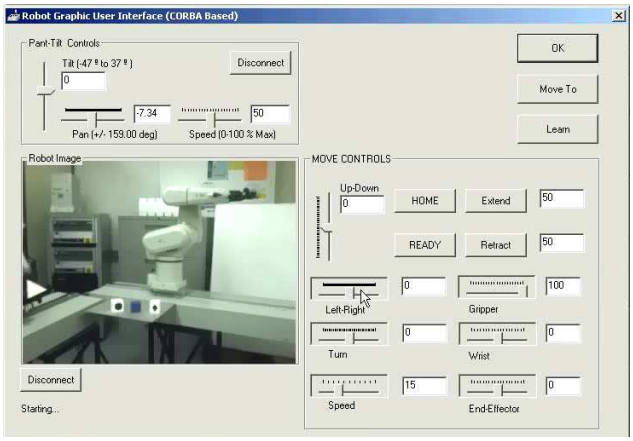
In figures 6.14, 6.15, the robot is moved down to reach an object.



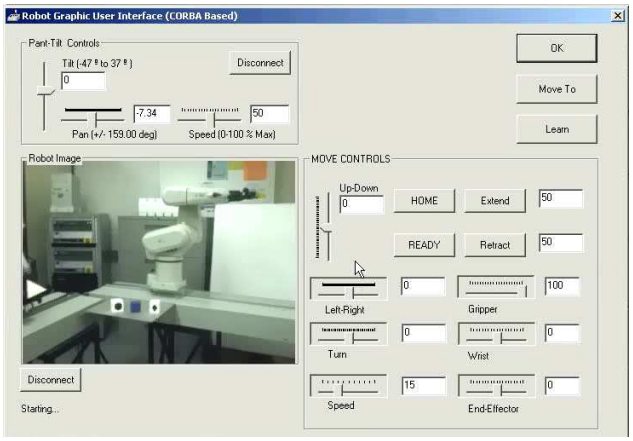
(a)Frame 514



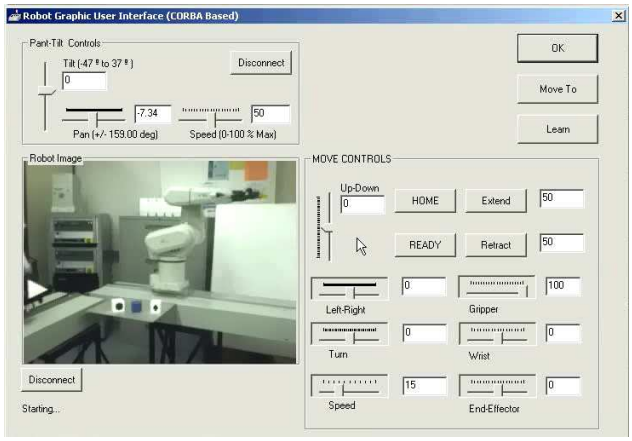
(b)Frame 587



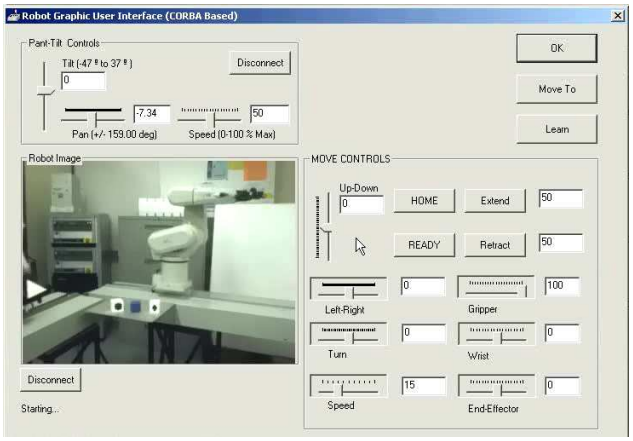
(c)Frame 589



(d)Frame 595

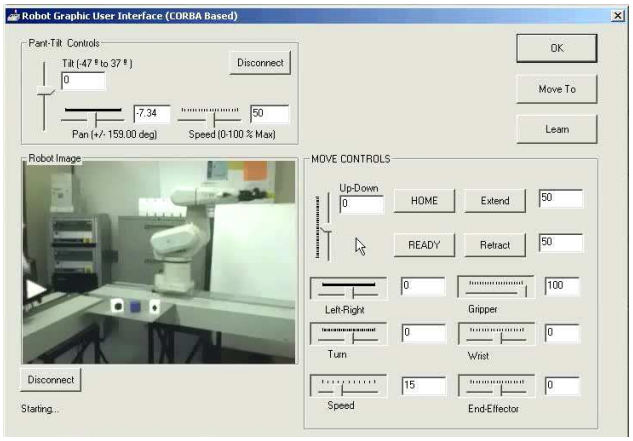


(e)Frame 619

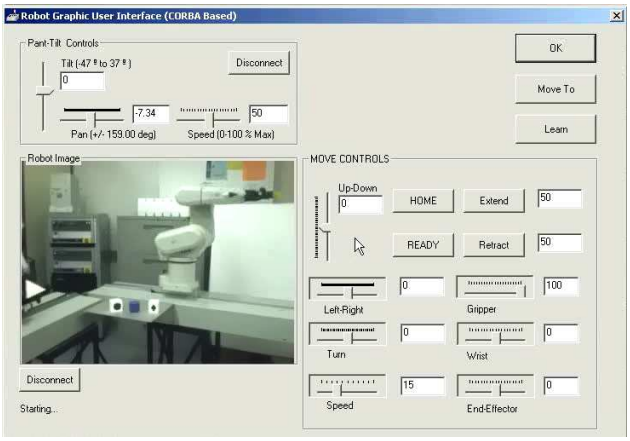


(f)Frame 635

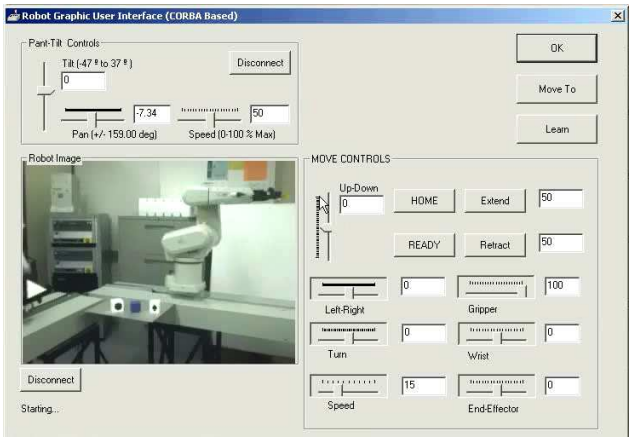
Figure 6.12: Moving the robot. Part-I. In this sequence, the robot controller receives a command to move to the right side from the point of view of the user.



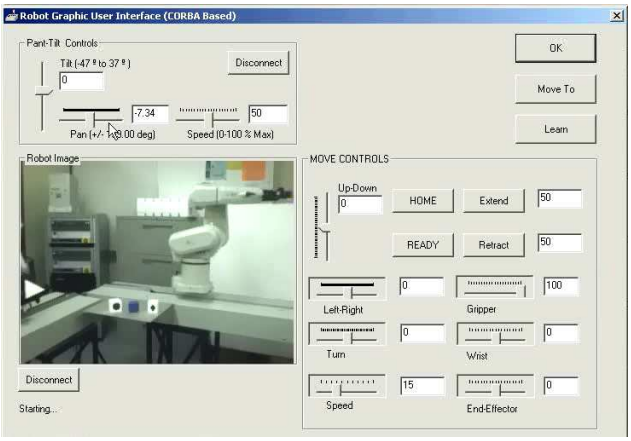
(a)Frame 648



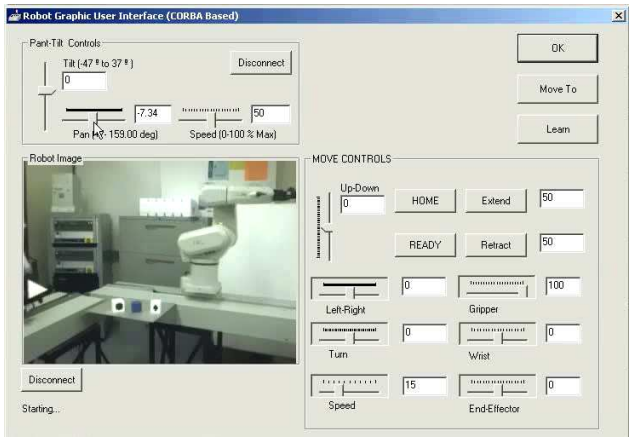
(b) Frame 664



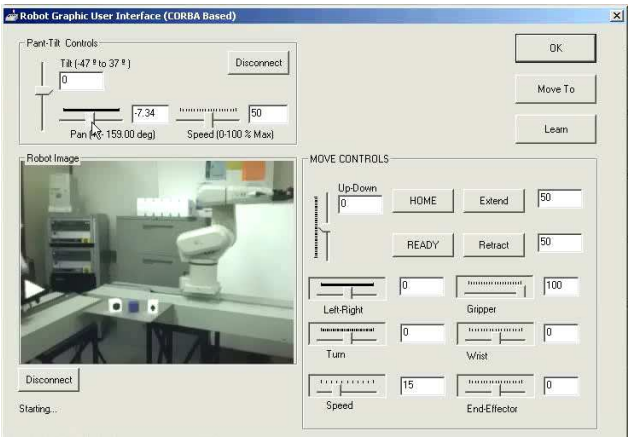
(c)Frame 678



(d) Frame 709

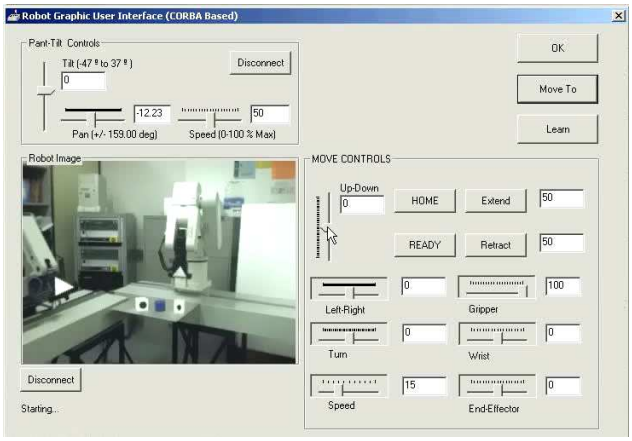


(e)Frame 737

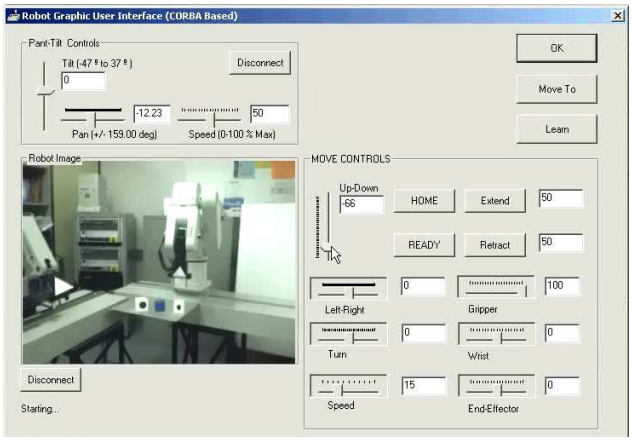


(f) Frame 752

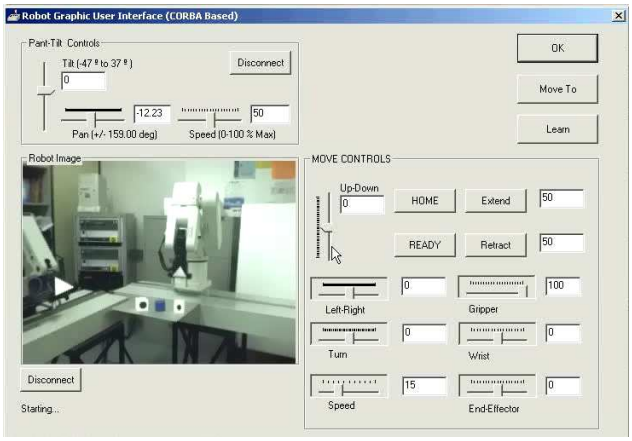
Figure 6.13: Moving the robot. Part-II. This sequence of images shows the total movement of the robot to the right side.



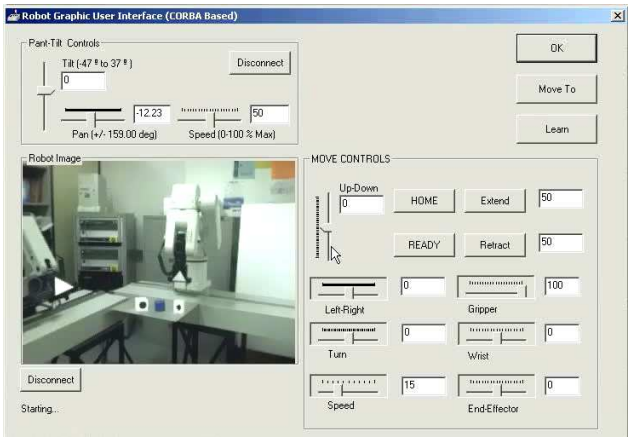
(a)Frame 1515



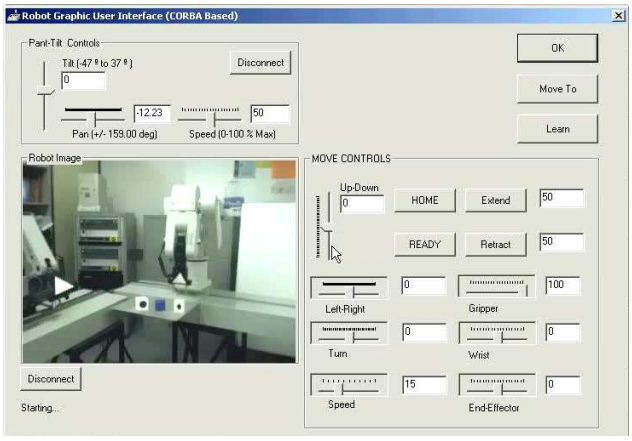
(b)Frame 1619



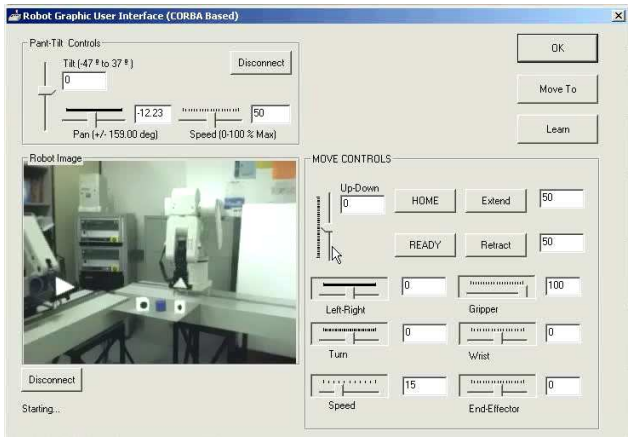
(c)Frame 1621



(d)Frame 1652

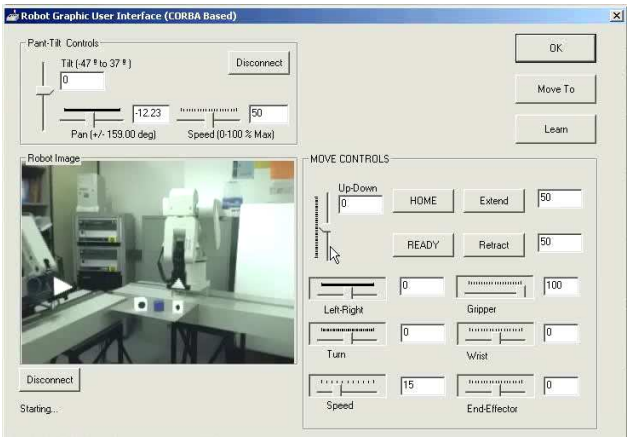


(e)Frame 1667

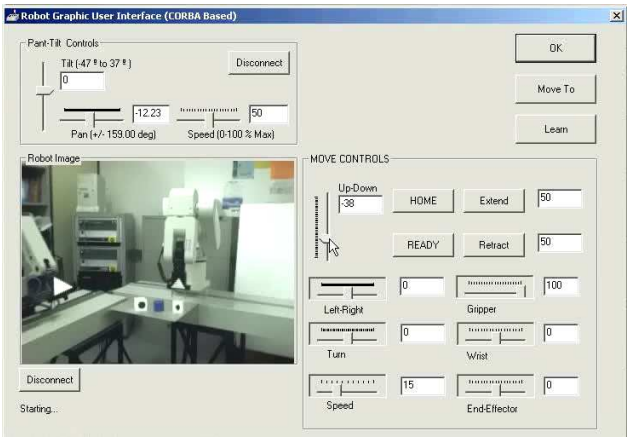


(f)Frame 1680

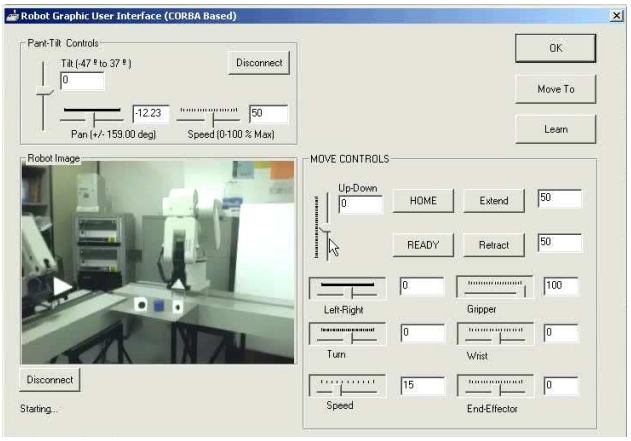
Figure 6.14: Moving the robot. Part-III. In this sequence, the robot controller receives a command to move move the arm down. The images shown how the slide to move the robot arm up-down is pressed and depressed. A positive number indicates a UP movement, while a negative number is a DOWN movement.



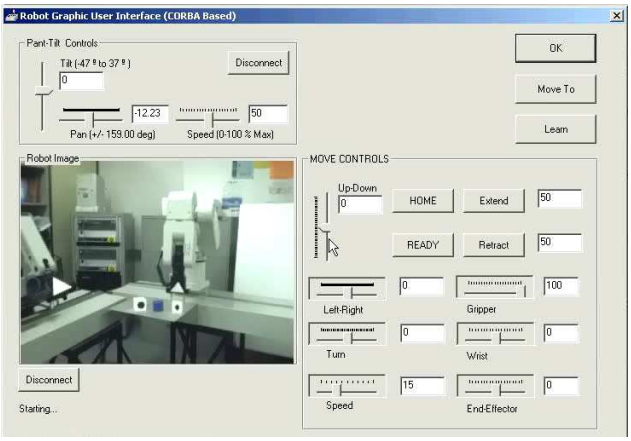
(a)Frame 1696



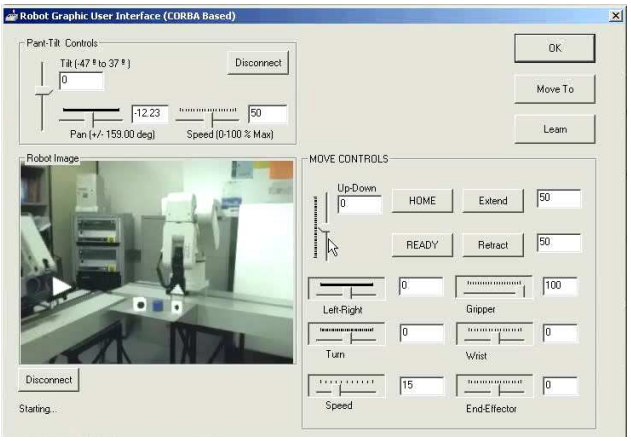
(b)Frame 1784



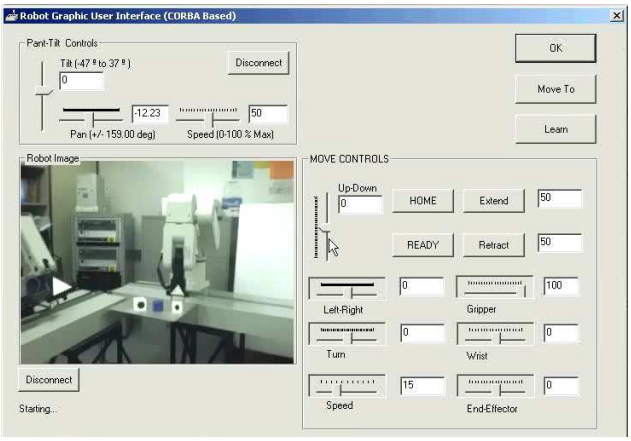
(c)Frame 1801



(d)Frame 1818



(e)Frame 1834



(f)Frame 1846

Figure 6.15: Moving the robot. Part-IV. In this sequence, the robot controller receives the last commands to reach an object moving down the arm.

6.2 Integration of a remote operated vision system

In previous system the image server was a single source of image stream that does not have any direct interaction with the user interface. The information flows through the event channel at specific rate. The user only needs to connect or disconnect from the channel, and more users can do the same action. Now, in order to improve the dexterity of the robotic system is necessary to increase the capabilities offered by the vision system to the wrapping level 3 shown in section 5.3.2.

Next level integration compounds the inclusion of three main vision capabilities: *extracting* objects' features, *finding* objects and *tracking* of them into the image stream. The first capability is executed as an off-line task, but it can be used in a remote mode or console mode.

In this experiment we want to test the capabilities of the vision system to be operated remotely. Then, the three previous capabilities will be tested using a small set of blocks with basic shapes. The vision system is composed of a stereo camera from videre design company. It has a frame rate of 10 images per second when using in color mode, although in each sample is sending a pair of images (left and right). The resolution of these images is 320×240 pixels in RGB color.

6.2.1 Recognizing objects

There are many ways to extract the visual properties of specific objects. Due to we will demonstrate how to manage the block-world problem, then the main objects are labeled blocks. These blocks are shape-labeled, it means each blocks has a specific shape marked in one or several sides. Each block is a white cube with a black geometric figure inside of it. This preparation is not by chance, we try to reduce the vision problems by providing color contrast in the images. Also, the images are processed in grey scale to reduce processing time. In this application there is a special object, this is the *end-effector* of the arm manipulator. This object is always the first object on the list of learned objects. The parameters computed for each object are: area, perimeter, Hu moments and a number of features (corners). Also a template image of the object is saved. In order to select the object to learn we include another restriction on the object characteristics, it must be a solid object or solid shape. An interesting point in this function is that the user is selecting the object from the image stream provided by the vision server. To accomplish this function the user interface is prepared to send the LEARN (TRAIN) command when the user clicks inside of the selected object. In this way the vision server starts the learning process from the point selected by the user. The first step is a filling object algorithm that is shown to the user through the image

stream. Due to there are many sources of noise in the image, the training process uses an average of 10 processed images frames. At the end the user can accept or reject the parameter set for the object and he can assign a name to the learned object if it is accepted. If the object name already exists the vision system creates a second parameter set for this object. The vision system does not check if the object learned has similar characteristics of a previous one. This is a decision of the user. In the following figures 6.16, 6.17 and 6.18 it is shown a sequence for the learning object process.

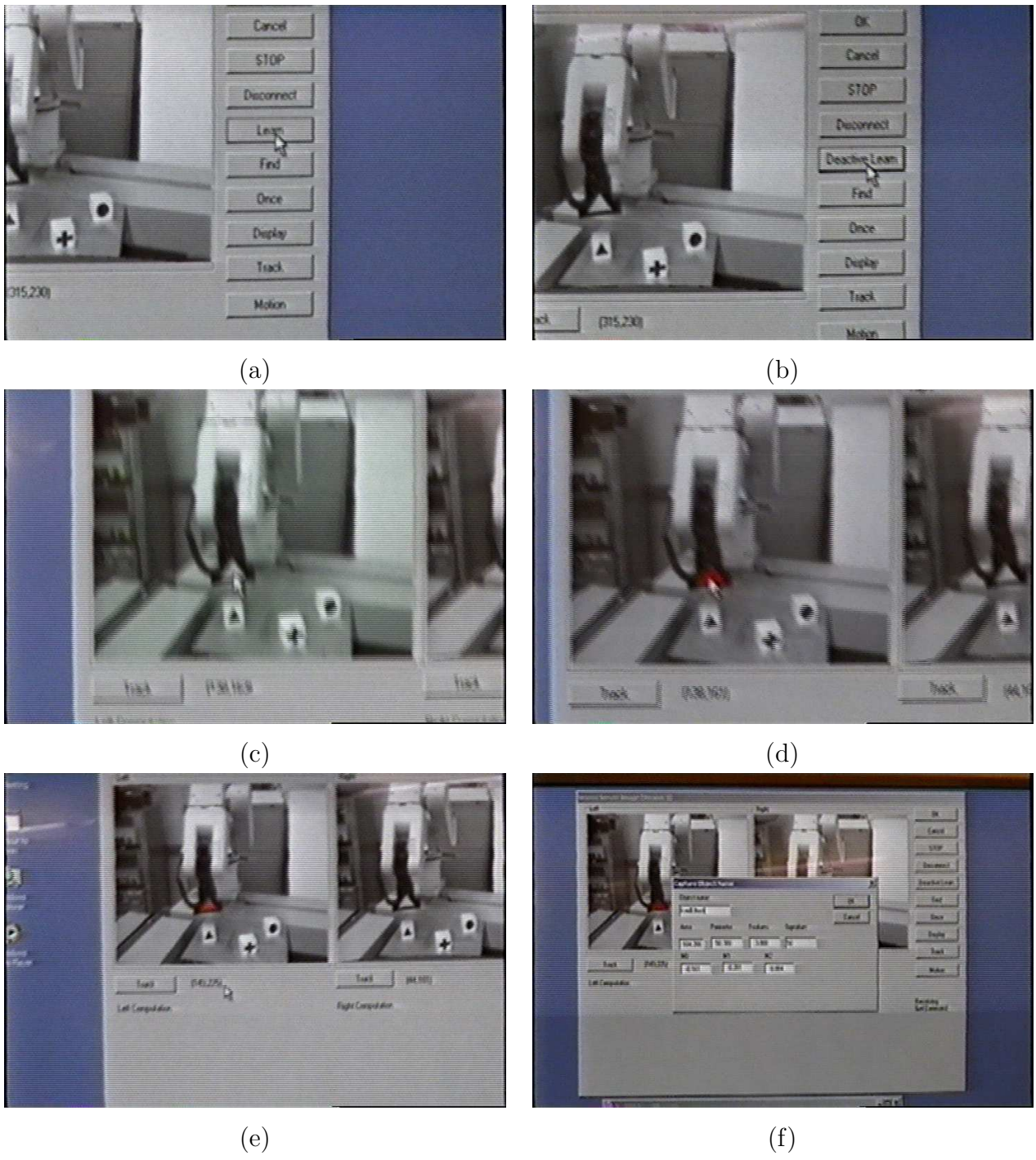
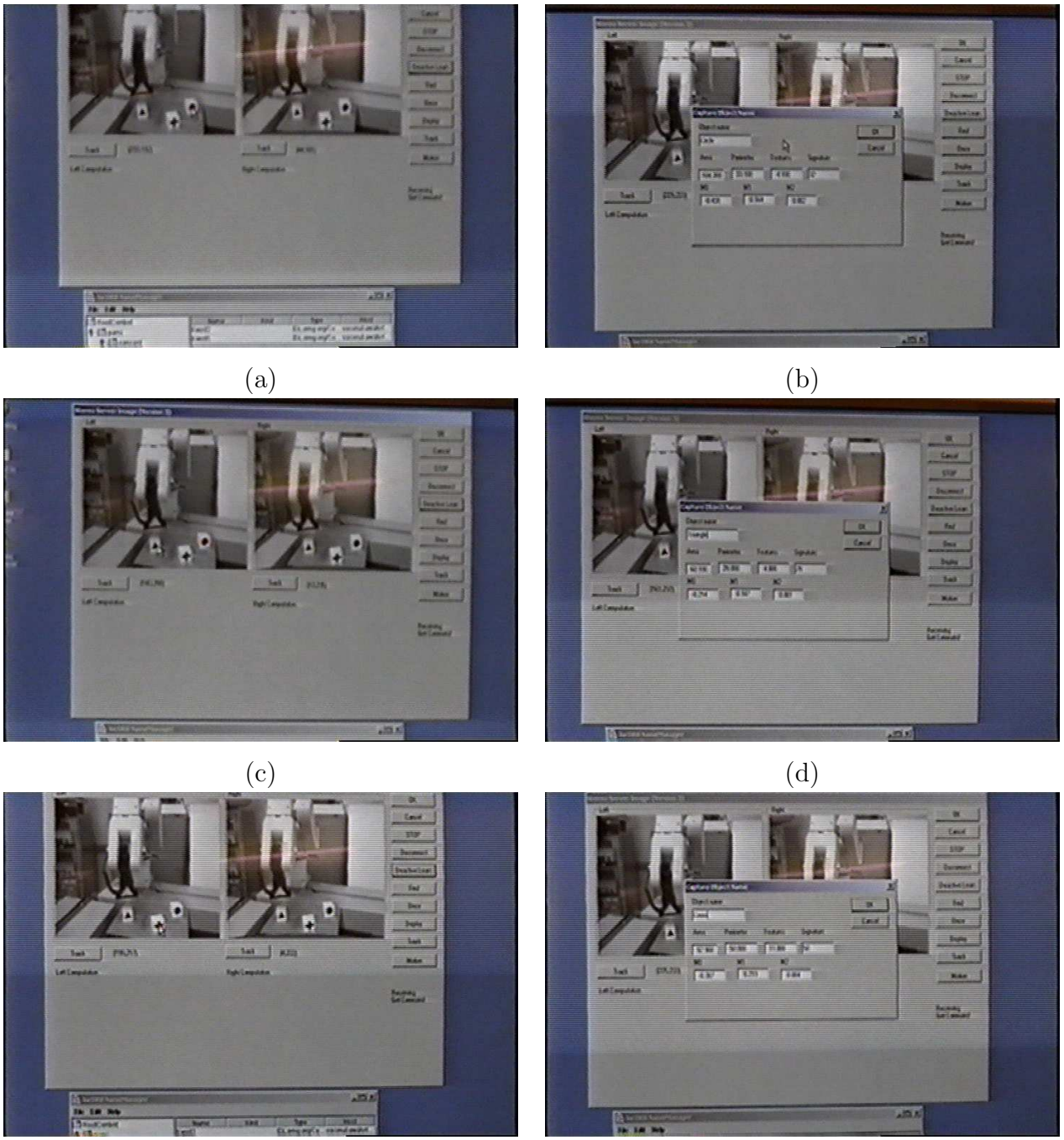


Figure 6.16: TRAINING command execution, Part-I. In (a) and (b) it is shown the option to activate and deactivate the learning (training) process. Once the learning mode is activated the user can select the object to learn by clicking inside the object (c). In (d) the selected object change to red color indicating that the system is gathering data to obtain object parameters. Object parameters are obtained in two steps indicated by fulfilling the object area (d) and next by coloring the border (e). Each step process 10 images in order to filter noise. Once the parameters are processed, they are displayed in a small window and an object name is requested (f).

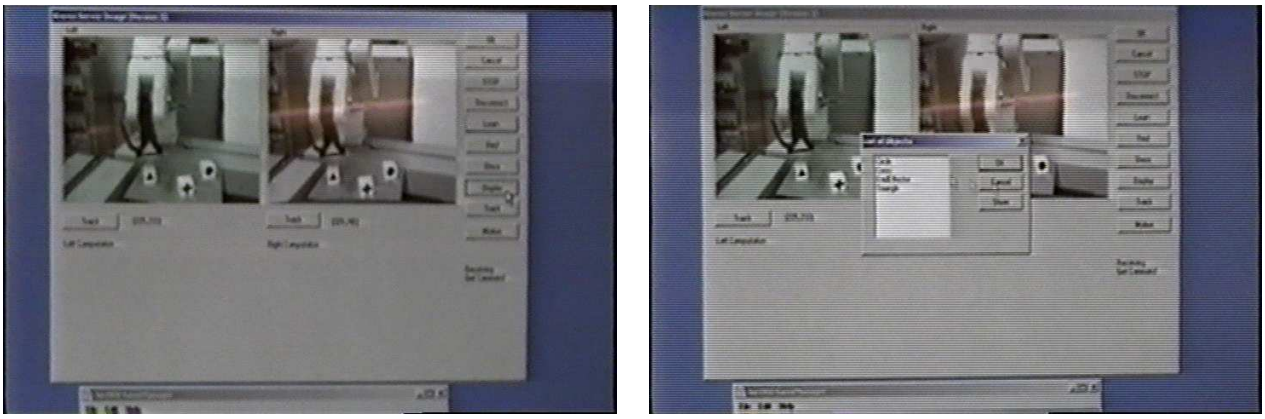


(a) (b)

(c) (d)

(e) (f)

Figure 6.17: TRAINING command execution, Part-II. In the following sequence more objects are learned. In (a) the block with a circle is selected and its parameters are shown in (b). In (c) the block with a triangle is selected and its parameters are shown in (d). In (e) the block with a cross is selected and its parameters are shown in (f).



(a) (b)

Figure 6.18: TRAINING command execution, Part-III. The user can request for a list of learned object. In (a) the option to request for the list is selected. In (b) a small window with the list of objects is shown.

6.2.2 Finding objects

Once a set of objects is learned and their parameters are saved into a database, the next step is to provide a capability for finding them (object recognition). This function is executed for each object in three stages:

- First, a complete search in all image is realized using *template matching* method. In this case the template is positioned at the upper-left corner and it is moved through the image making a difference computation at pixel level. Appendix C explains in more detail what parameters can be adjusted to increased the computation speed or resolution performance. In the first stage a list of points with a lower value are stored depending on certain threshold.

- Second, the list is processed to eliminate all neighbor points that correspond to the same object. An Euclidian distance measure is computed and if this data is lower than a specific threshold one of the two pair of points is discarded.

- Third, the final object is selected using a weighted discrimination algorithm. For this procedure, each characteristic of the objects is compared with the corresponding parameter saved in the database. The difference is weighted according to the object to find. At the end, the object with the minimum difference and below a threshold is selected.

If no points are selected after the third step then the object cannot be found. If one point is selected, then this point represents the closer-to-center position of the object

into the image. This point is shown as a cross mark in the object. Figure 6.19 shows this process.

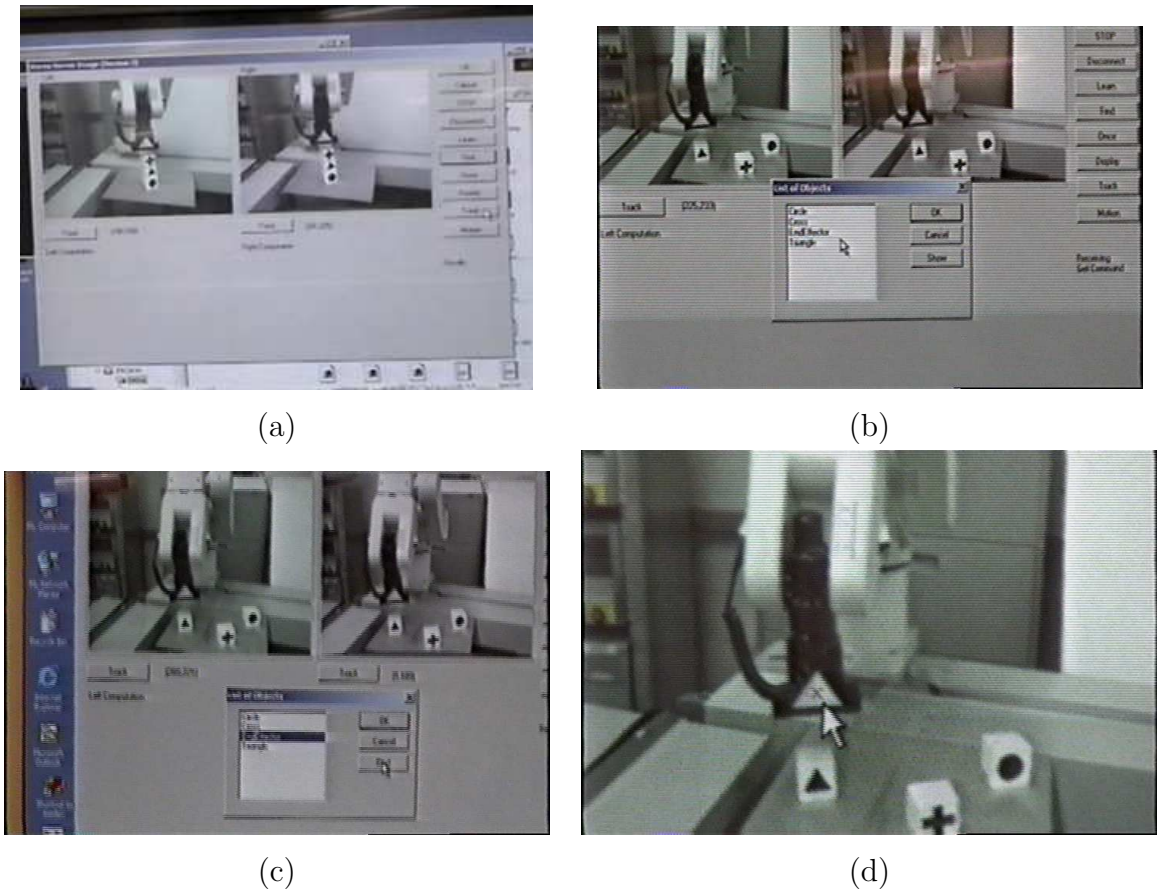


Figure 6.19: Finding object process. For this process the FIND option is selected (a). In (b) a list of learned objects is shown. The user select the object to find, in this case the End-Effector (c). Finally, after a short time the object is found and a cross mark is shown inside the object (d).

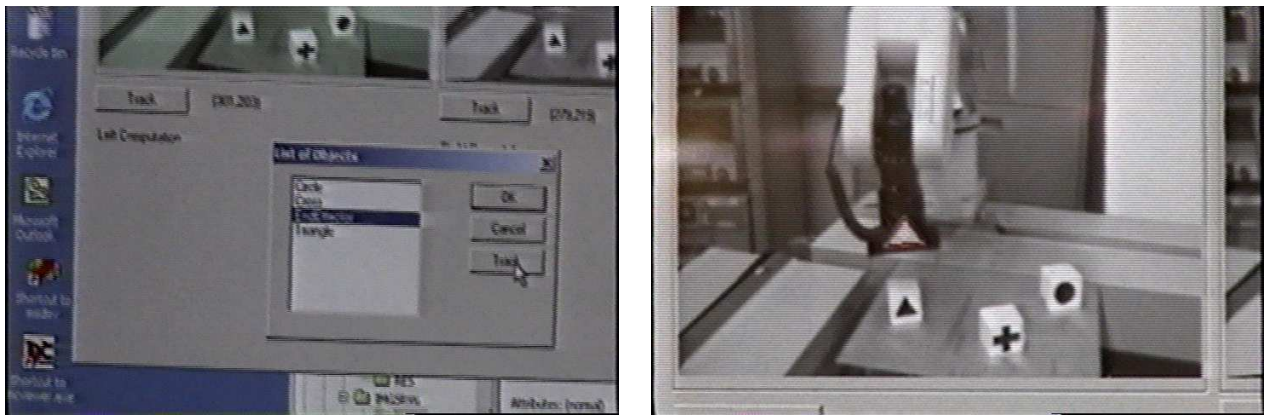
6.2.3 Tracking multiple objects

Finding one object is a computational expensive task. Finding several objects and tracking them could be more expensive. To deal with this problem the tracking function is executed in two stages:

- First, for each object a complete search is realized to find its center position. This is done using the previous finding command.
- Second, using the center point of the object the next search is reduced using a different heuristic. Basically, the search is started from previous position and using

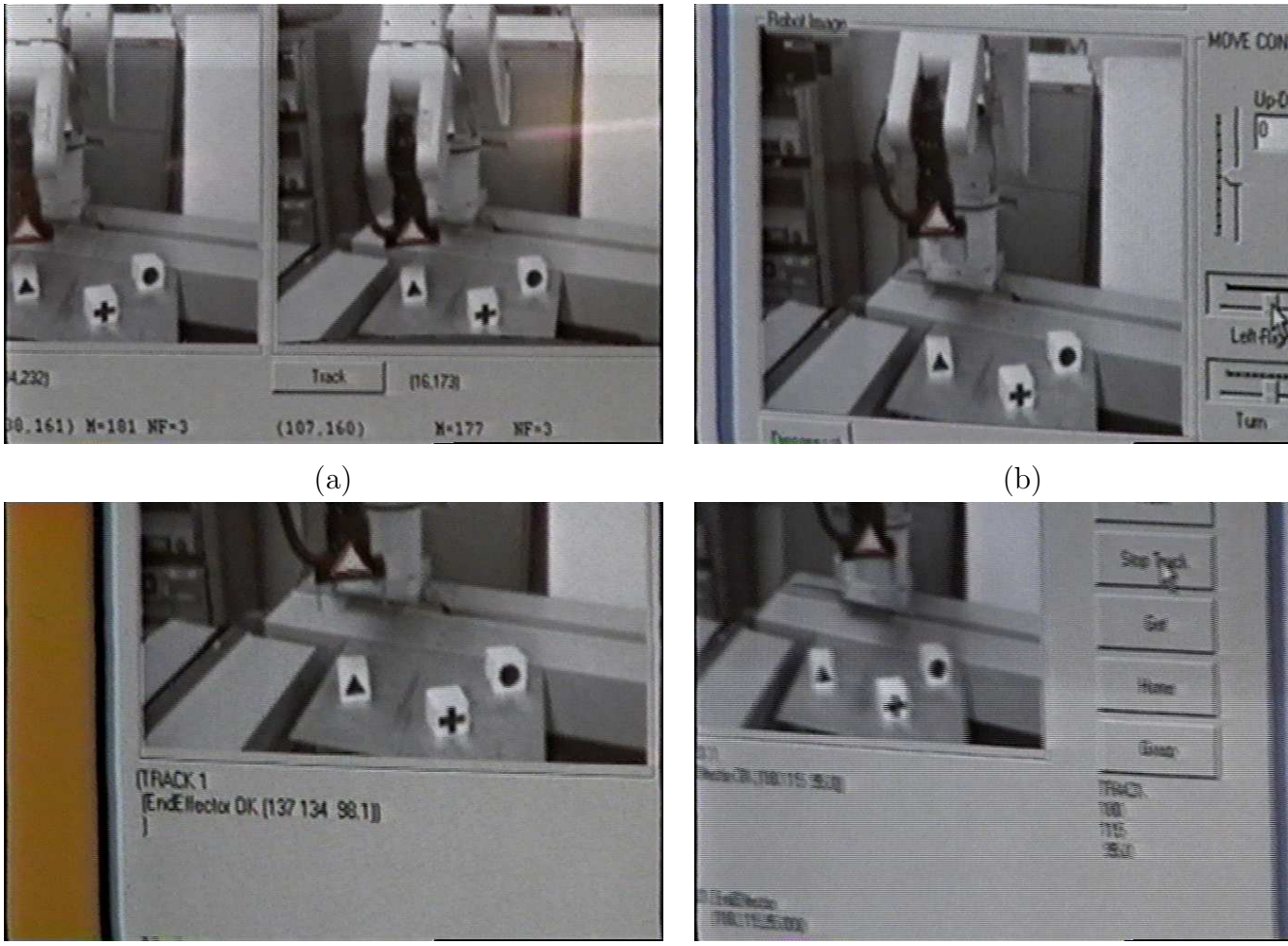
the interline method [Guedea *et al.*, 2003b]. This method follows the contour of the object and recognize some invariant and features. Basically, it tracks the next center position of the object and uses some heuristics to determine if the object is valid. Usually, with some changes on the lighting the object can be lost for one frame and recovered in the next frame.

A searching list is built in order to keep tracking of multiple objects, including the end-effector. Due to the vision system is a stereo system, two tracking procedures are performed, one on each image frame (left, right). Once both objects (left and right) are tracked, a depth distance is computed. This distance is calculated from the center of the camera to the center of the object.



(a) (b)

Figure 6.20: Tracking object process, Part-I. The system has a list of objects to track where the first object is by default the End-Effector if it must be tracked (a). Then a chaining list is used to keep information about the last central point registered for each object. In each subsequent frame the object is located using the interline method (b), the vision system mark the center position with a + sign. Furthermore, it indicates which is the contour pursued.



(a) (b)

(c) (d)

Figure 6.21: Tracking object process, Part-II. In the following sequence the End-Effector has an initial position marked in the bottom of the figure (a). Next on (b) the robot arm is moved using a vertical slide. In (c) the new position is displayed on the remote interface. Finally in (d) the STOP-TRACK option is selected.

6.2.4 General operation and communication scheme

With the previous capabilities, the vision server is now ready to provide the services to any client in the network. These services are provided by using the image channel aforementioned (Section 6.1) and by using the interface defined in Section 5.3.2 (Figure 5.8). Now, another channel is added to send periodic information about finding and tracking processes, which are executed through commands sent by the clients, Figure 6.22 shows these communication channels. This communication approach provides a loosely coupled method which facilitates the design of inter-connected components.
Due to the different services provided by the vision system there are several operation modes and some of them are exclusive. Basically we have the following modes:

CONNECT The vision system is connected to both channels: Image Channel, where only raw and compressed images frames are sent and Data Channel where the information about some commands (FIND and TRACK) is answered.

DISCONNECT The vision system stops sending information through the previous channels.

LEARNING In this mode the vision system waits for an initial position to start a learning process. The image is changed from color to grey scale mode.

FINDING In this mode the vision system is searching a specific object. If the object is found the system returns its center position on pixels.

TRACKING In this mode the vision system periodically tracks a list of objects. The searching time depends on the size of the list. The data returned is a list of objects with its track situation (OK, FAIL) and its center position if the object is OK.

In Figure 6.23 there is a graph explaining these modes. CONNECT and DISCONNECT modes are exclusive. CONNECT mode can coexist with the other modes. LEARNING, FINDING and TRACKING modes are exclusive, only one of them is active at one time. These operation modes are configured in this way to avoid multiple functions on the server side. Although this configuration could limit the interaction with several clients, by using ORB the information could be distributed through the Event-Channels as it is shown in Figure 6.24.

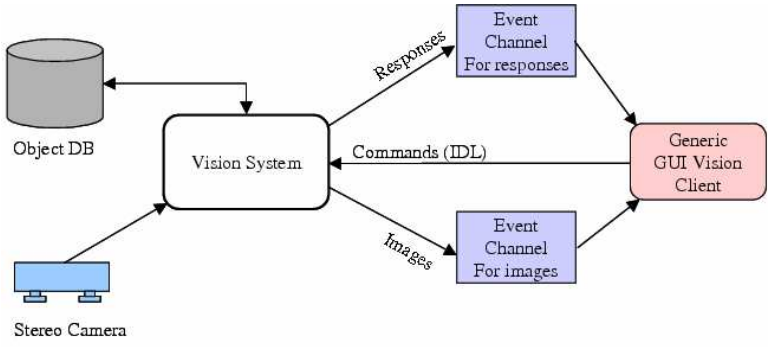


Figure 6.22: Vision Server communication outline.

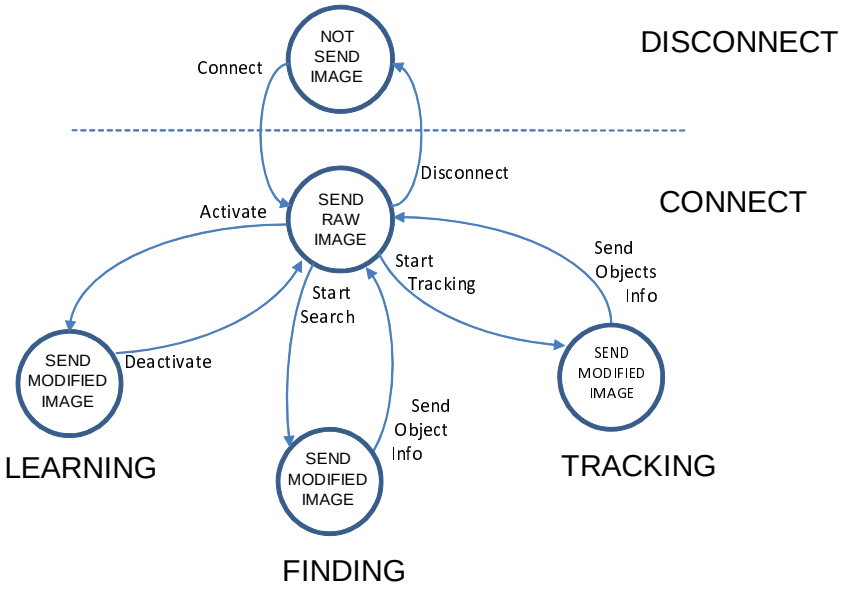


Figure 6.23: Operation Modes for the Vision Server.

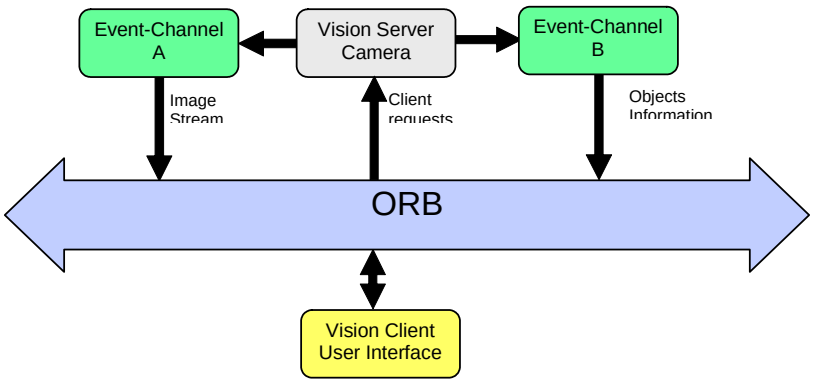


Figure 6.24: Vision Server (wrapper component) with CORBA Event Services and the user interface.

6.3 Integration of an autonomous distributed robotic system

This work was presented at [Guedea *et al.*, 2006a]. This is the final goal that we are pursuing to enhance the behavior of a robotic system. Mainly, we want to improve the dexterity of a robot by adding more "intelligent" components. But this addition is not simple or easy, each component has its own input and outputs and it is necessary to adapt them in a comprehensive manner. In previous sections we explained how to incorporate a robot arm remote manipulation with image surveillance and then how a vision server with capabilities for object recognition and tracking objects was developed. Now we will show how all these components and a new component, *planning server*, are integrated to manage the block-world problem. The main components are shown in Figure 6.25. In our design we decide to create a component which is in charge of orchestrate all activities. This component is named "Coordinator", in previous papers was called "brain controller" [Guedea *et al.*, 2006a] or "task controller" [Guedea *et al.*, 2004]. This component or module will be described in detail in the following sections.

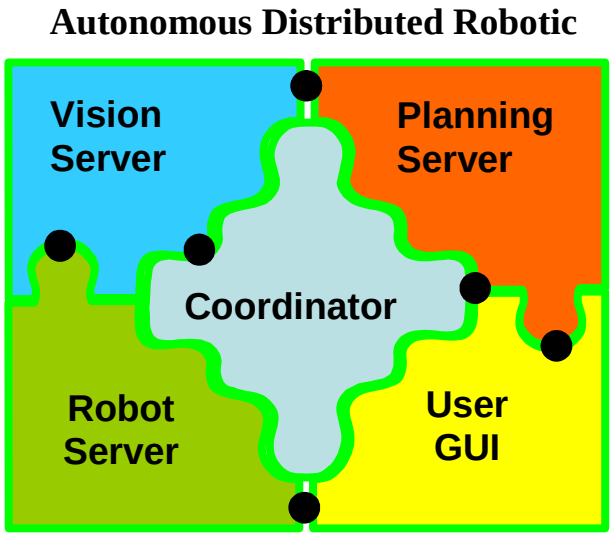


Figure 6.25: Main wrapper components to built the autonomous robotic system.

6.3.1 Off-line tasks

Before remote operation of the robot arm manipulator could be started some off-line tasks must be executed. First, we must "teach" the system about the objects to manage. This activity can be realized using a *Console-Server* or *Client-Server* approach. In the first case the vision system provides a GUI with two images (left, right) and a set of button functions. In the second case only left image is displayed and basic button functions are provided. Second task corresponds to the setup and configuration of the planner server. This is a console-server interaction where the user defines which actions, objects, and states will be needed in order to create a sequence of operations to accomplish the goal. The last task is to provide an ordered servers activation. This must be done using the Naming Service from CORBA specification.

Learning objects

In order to manipulate blocks, first we need to learn the visual characteristics of these objects. In our block-world there are four shapes and the End-Effector. Basic shapes are one black circle, one black triangle, one black cross, and one black square. The End-Effector has a white triangle shape. Figure 6.26 shows three of these objects and the End-Effector in their initial position.

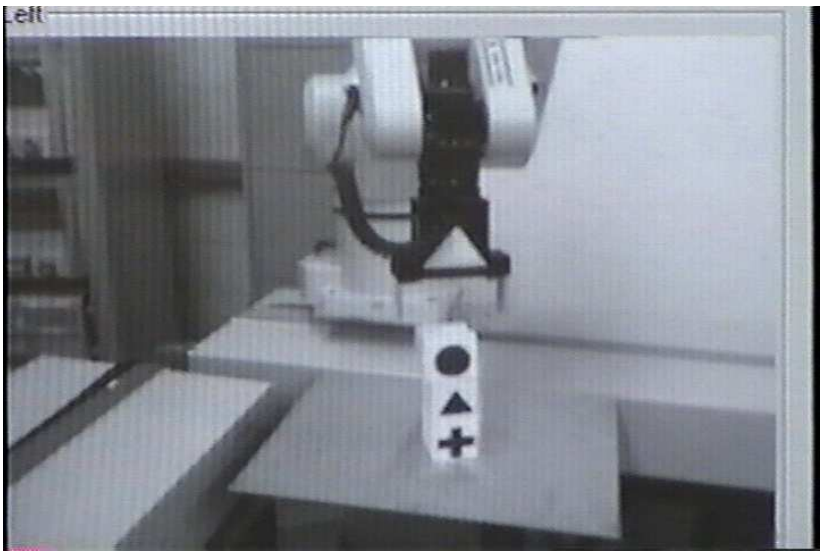


Figure 6.26: Main objects used in the block-world problem.

Configuring operations, objects and states into the planner

The planning server is a general purpose planner. In this sense, it needs to be setup for each planning scenario. This implies to define the set of operations (actions), the set of objects to which operations apply, and the set of possible states for each object. In this application, the operations are related to the activities realized by the robot arm manipulator. The set of objects is the same set of blocks learnt by the vision system, including the End-Effector. Objects states are related to the relative position among blocks, End-Effector and some environment elements, such as the table where the blocks are placed, also there are some status related to End-Effector. Instead of using the same set of commands defined in the IDL robot interface, a subset of macro operations is defined. Table 6.5 shows a summary of these configurations and the possible states for some objects.

Table 6.5: Operations defined for the block-world problem

Operation	**Parameters**	**Preconds**	**Effects**
Pick-Up	(< *obj* > OBJECT)	(clear < *obj* >) (on-table < *obj* >) (arm-empty)	(holding < *obj* >)
Put-Down	(< *obj* > OBJECT)	(holding < *obj* >)	(clear < *obj* >) (arm-empty) (on-table < *obj* >)
Stack	(< *obj*1 > OBJECT) (< *obj*2 > OBJECT)	(clear < *obj*2 >) (holding < *obj*1 >)	(arm-empty) (clear < *obj*1 >) (on < *obj*1 > < *obj*2 >)
Unstack	(< *obj*1 > OBJECT) (< *obj*2 > OBJECT)	(on < *obj*1 > < *obj*2 >) (clear < *obj*1 >) (arm-empty)	(holding < *obj*1 >) (clear < *obj*2 >)

Servers sequence activation

In this configuration several servers are need. In order to avoid IOR's file transferring, a Naming Service is used, together with several Event services. Figure 6.27 shows this sequence. The first server to be activated must be the Naming Server. Next, the event services for vision services are activated and they register on the Naming server. Following these activations is the vision system, which connects its outputs for raw image transfer and image information responses. Finally, robot server and planner are started and their IOR references are registered. The coordinator is not a server itself. It is a client for the servers and it works like a "bridge" between the user and the robotic system. The last servers (vision, robot, and planner) can be started in any sequence after the Naming and Event servers but must be ready before the coordinator program is launched.

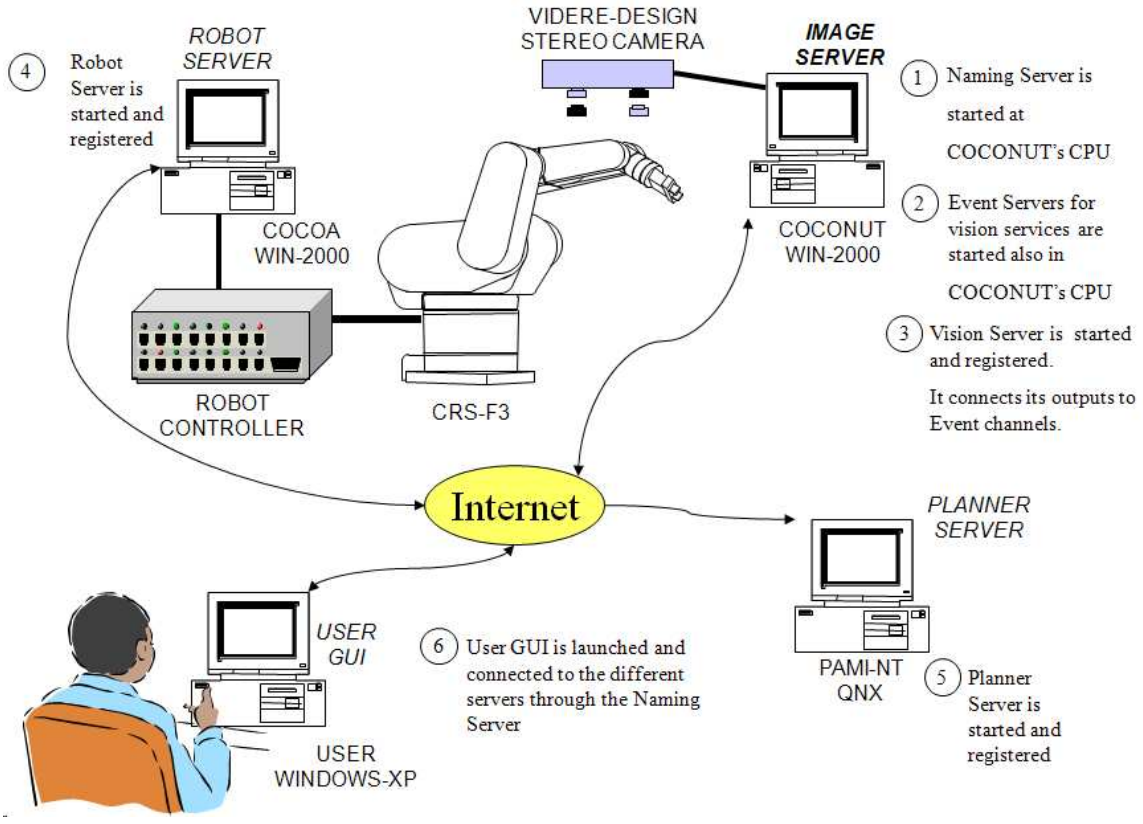


Figure 6.27: Starting sequence for an Autonomous Remote Commanded Robot. IOR references from event services, and servers are registered in the Naming server. At the end the user graphics interface gets the IOR references from this source to communicate with the different entities.

6.3.2 On-line operation

In this robotic application the main goal of the system is to arrange a set of blocks according to user instruction. This instruction is provided through a command line window where the user establishes the goal using a LISP format. Once this instruction is received into the coordinator module, it starts a sequence of activities to accomplish the task. In order to manipulate blocks, first its necessary to identify each block's position in the current situation or configuration. Second, the actual physical position information is converted into information that the planner module can understand. Physical location information is abstracted as which block is on top of which block. Based on these facts, the planner module is asked to generate a plan, i.e a sequence of actions that the manipulator has to follow. As it was mentioned all these steps are orchestrated by the coordinator module, which queries the block position, asks parsing and conversion, requests plan generation, and commands the manipulator to move it at the same time that it is receiving visual feedback.

Communication schema

As shown in Figure 6.28, The coordinator has two event channel connections and one IDL interface connection with the vision server, one interface with robot server, and one interface with planning server. Thus, it can initiates blocks manipulation by asking vision information, existing blocks and their positions. Then using block class member functions, it processes the low-level vision information, requests planning operation to planning server. After receiving planned actions from the planning server, the coordinator sequentially sends the commands to the robot server. As the robot server moves the block, the coordinator checks if each operation goes correctly.

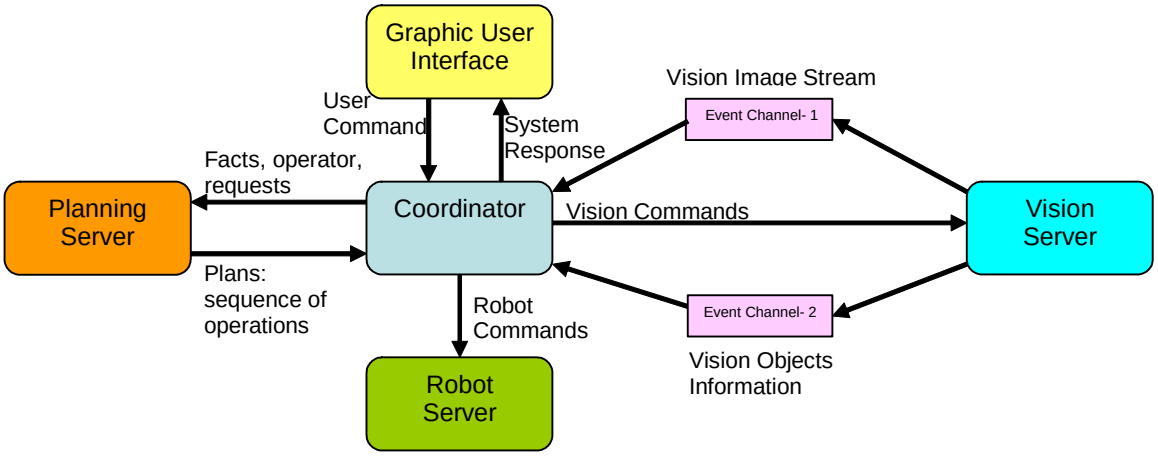


Figure 6.28: Coordinator connections with other modules or wrapper components.

General Operation

To start the system operation the user must send a command to achieve a specific goal, i.e. a final block arrangement. Figure 6.29 shows an example of this user instruction. This command is received by the coordinator which starts the following sequence of operations:

a) It asks to the vision server for the objects stored in its database. To performs this operation a GET_OBJECTS command is sent. See Figure 6.30 (a).

b) If the objects are in the database then it asks to the vision server for "FIND" these objects in the current image stream. See Figure 6.30 (b).

c) If all objects are found then it is possible to generate a plan (step d), if not, then the coordinator responds to the user that there is no way to achieve the goal. See Figure 6.31.

d) With the information received from the "FIND" command, the coordinator converts (parsing) this information and makes the data fit to GPLAN IDL interface. In other words, it sends the facts taken from the vision system to the planning server. See Figure 6.32 (a).

e) With the current facts and operators in the planning server, and also the goal information, the coordinator requests to the planning server for a plan, i.e. a sequence of operations. See Figure 6.32 (b).

f) The planner server can respond with two answers: a) there is a feasible plan or b) there is not plan to get the final goal from current situation and operators. In the first case, the next operation is described in step (g), for the second case, the coordinator responds to the user that it is not possible to achieve the goal from current situation. See Figure 6.33.

g) Once all objects are found and there is a plan to achieve the goal, the coordinator requests to the vision server for "TRACK" specific objects according to the current operation and also commands the robot to execute a sequence of commands. These commands are the services provided by the robot server. Next section explains how these commands are executed. See Figure 6.34.

h) Once the current operation is realized, the coordinator ask the planner server for the next operation. Once all operations are realized the autonomous robotic system completes the goal, and the coordinator sends a responds to the user that the goal was achieved.

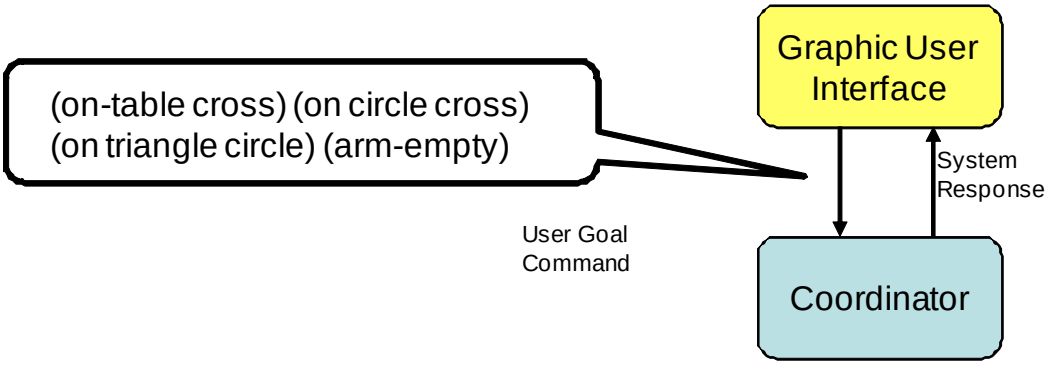
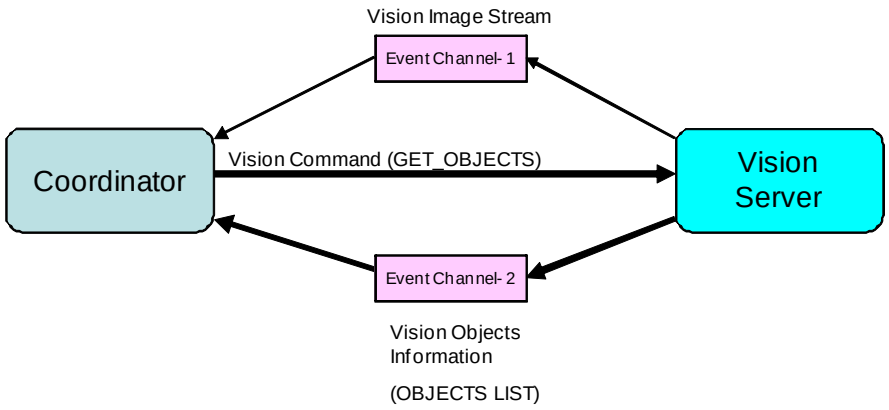
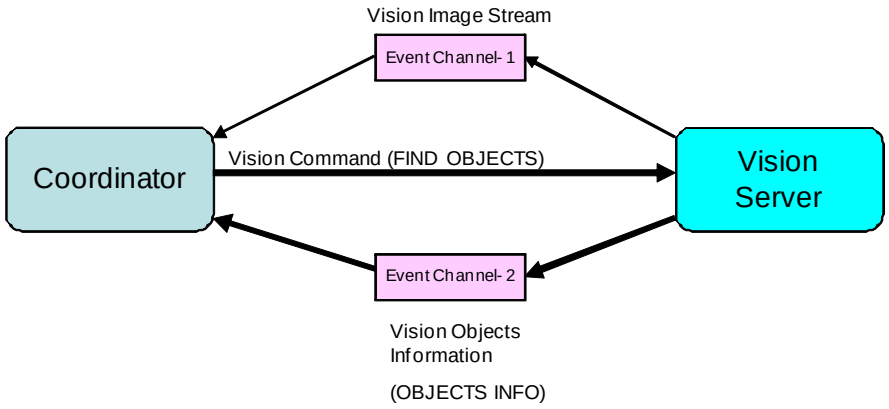


Figure 6.29: User goal command sent to the coordinator module.



(a)



(b)

Figure 6.30: Coordinator sends two commands to the Vision Server. In (a) it is asking for the object's list. In (b) it is asking for the position of each object.

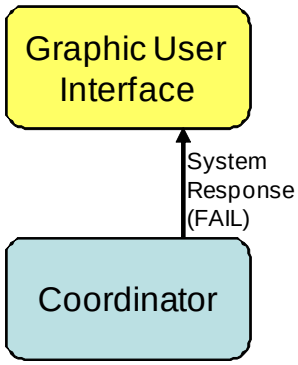


Figure 6.31: Coordinator responses a failure to get the user goal due to it is not possible to locate all objects.

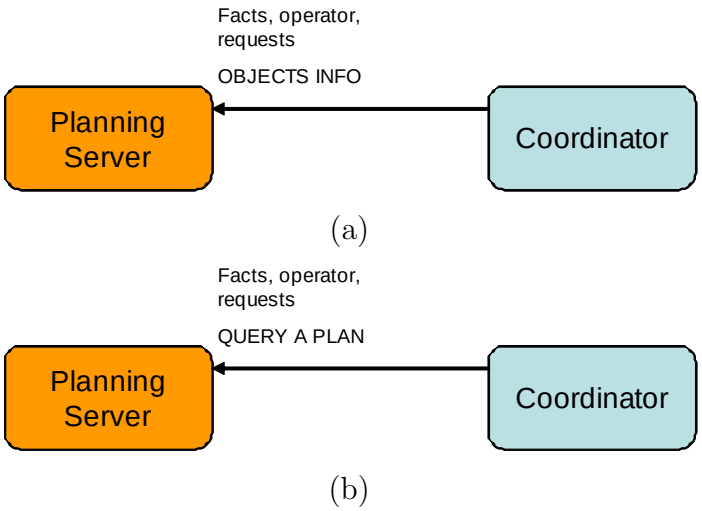


Figure 6.32: Coordinator sends information to the planner and asks for a plan. In (a) it is sending the object information. In (b) it is requesting for plan to achieve a specific goal.

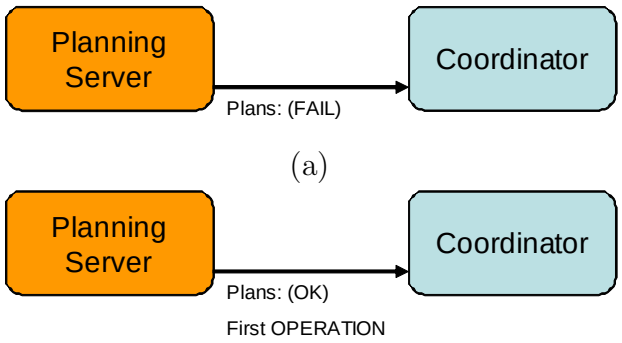


Figure 6.33: The planner server responds to the coordinator. In (a) it is impossible to achieve the user goal, a FAIL response is sent. In (b) it is possible to achieve the goal and the first operation of the plan is sent.

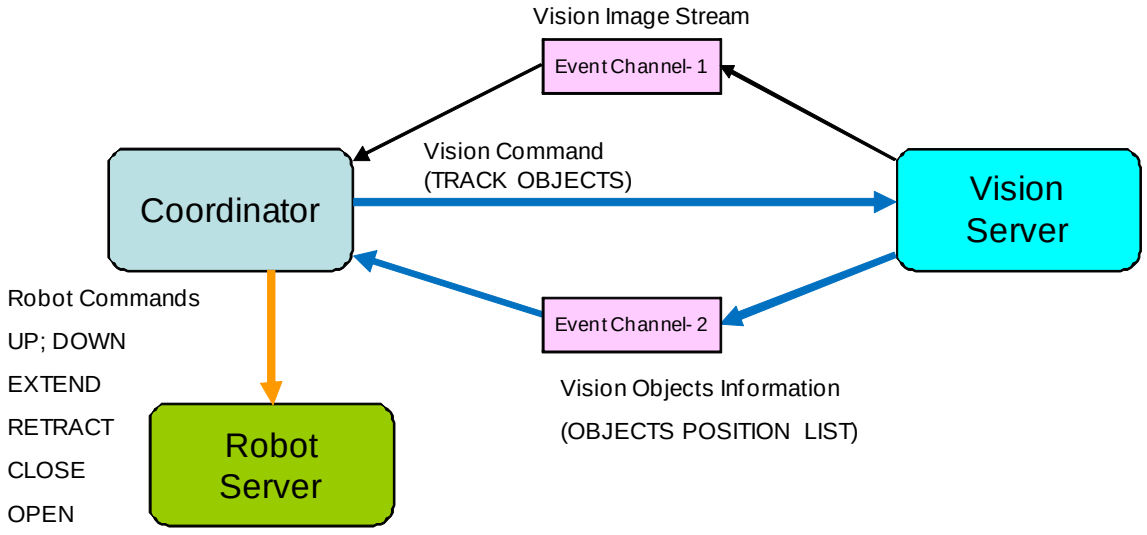


Figure 6.34: Coordinator is executing an operation. This execution is performed in an initial stage, and next in a cyclic manner. First, the coordinator asks to the vision system for "TRACKING" a set of objects. Usually two objects are tracked at the same time. Next, according to the information provided by the vision system, the coordinator moves the robot arm using a set of basic robot commands. Not all commands are shown in the figure.

6.3.3 Robot Server Commands

The operations generated by the planning server are not direct commands to the robot server. The coordinator must translate these operations in a sequence of micro-tasks for the robot server. These micro-tasks are not fixed or tied to a specific planning operation. They vary according to the current information sent by the vision server. It is necessary to have a control scheme to select the right robot command according to the position of each object.
Most of planner operations are converted to basic operations such as GRASP, OPEN and CLOSE micro-tasks. Let be the current micro-task "GRASP Rectangle", then the coordinator must do two basic operations:

a) Align End-effector with the Rectangle object, and

b) Close the gripper.

We designed a simple control schema where an approximation method is used to align the position of the objects. Figure 6.35 shows this idea. The process to align two objects is realized according to five states (0 to 4), the error measures in pixels, and the depth measure. The computed values are Err_X, Err_Y, and Err_D for depth difference.

State 0 Due to it is difficult to estimate if there is a blockage from the current position of the End-Effector to the position of the "Rectangle", then a first movement is to go UP if the End-Effector is below a minimum distance. This movement corresponds to state 0 (initial).

State 1 From a top position the coordinator tries to align first the objects in the pixel axis x.

State 2 The coordinator tries to align objects according to depth measure.

State 3 The coordinator makes an approaching in pixel axis y going down.

State 4 In all previous states the coordinator makes a raw approximation (high gain), between the two objects using vision feedback information. In this state a finest alignment is done (low gain). Once the distance between the objects center is below a threshold the alignment is finished.

In state 4 there is a validation to check if the movement to close Err_X is done using TURN command or MOVE_HR command. TURN command gives a larger grained movement than MOVE_HR. Also, if for some reason the object is moved the coordinator will compensate the error accordingly.

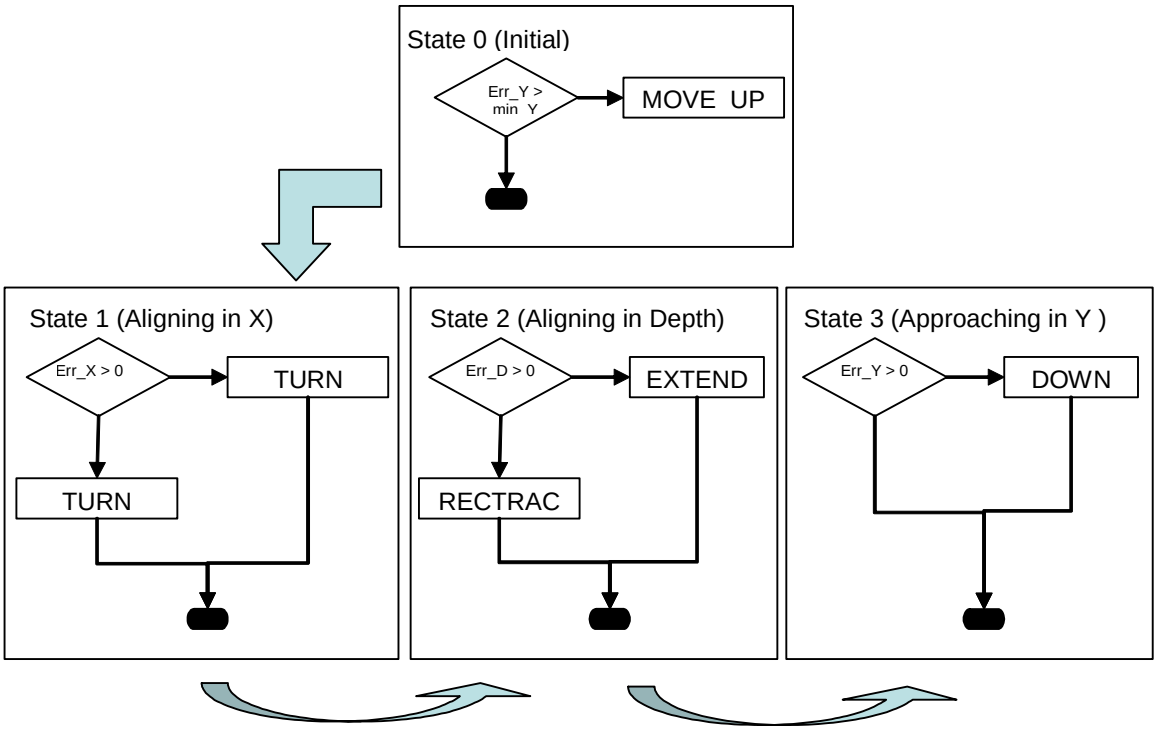


Figure 6.35: First states to approach the End-Effector or object position relative to other object.

6.3.4 Integral Example: Grasping an object

In previous sections a detailed explanation about how to setup the vision system, the planner and how to command the robot was given. In this section a small example is described to visualize the information provided by the vision system while the coordinator is trying to grasp an object.

Figure 6.36 shows nine images for this example. First, in (a) the initial state of the system is shown. In (b) a tracking command for the circle block is issued. This is proof by the white cross in the middle of the circle. Next, in (c) a tracking command for the End-Effector is issued. Before starting a tracking, a find operation is executed and due to tracking and finding operations are mutually exclusive the system stops previous tracking. In (d) both objects End-effector and circle block are tracked. In (e) the robot arm starts the alignment process but due to both objects are very close in axis x and depth measure there is not a significant movement among them. In (f) the robot arm is moving down, it is in state 3 closing the gap on the axis y. A very small movement can be observed comparing the line on the robot base against the End-Effector. In (g) the coordinator gets into the state 4 where the movements are very small. In (h) the coordinator stops the tracking due to all errors are below a specific limit threshold. Finally in (i) the coordinator commands the robot to "close" the gripper.

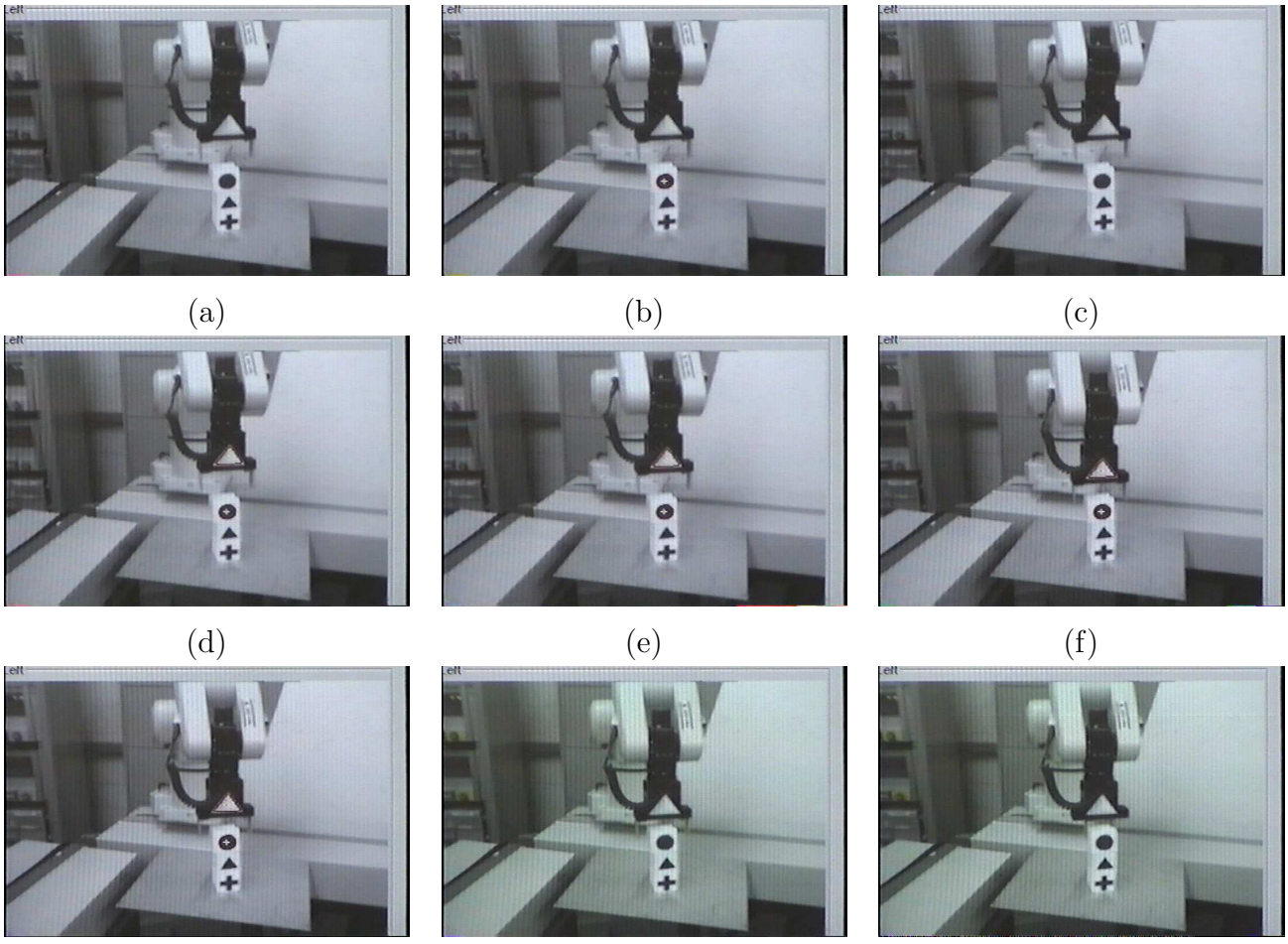


Figure 6.36: GRASP command execution. Initial state is shown in (a). Next a circle tracking command is issued in (b). In (c) there is a temporal suspension of the track process while the vision system is finding the End-Effector. In (d) both object are tracked. In (e) the alignment process is started. In (f) the robot is moving down. In (g) the system is in state 4. Stop tracking is shown in (h), while in (i) the coordinator ask the robot to close the gripper.

Proving dexterity of the robot for changes in the environment

The previous section shows the sequence followed for the system to grasp an object. In this case, only the robot arm is moving meanwhile the objects in the environment keep fixed or static. In the following Figures 6.37,6.38 a small sequence where the object to pursue is intentionally moved is presented. In the image Dr. Song's hand is moving the object. First using a small artifact and then using only the hand. As result, the measured distance on pixels between the center points of two objects is changing. These objects appear with a border line on them. Then, the robot arm is trying to adjust this distance by executing the corresponding movement command.

6.4 Results and Discussions

6.4.1 Remote Operated Robot

Moving robots remotely has some important aspects to take into account. First, the naming service and the event service must start first. Second, object implementations (servers) for each component are started. Third, when the servers are ready the Robot Graphic User Interface (RGUI) interface can be started. The robotic system is located at the Pattern Analysis and Machine Intelligence Laboratory (PAMI-Lab) at the University of Waterloo, Canada. The first test was to move the robots using a Local Area Network. This test worked fine, the vision feedback delay is not perceived by the operator. Only a movement delay is caused by the same arm manipulator controller when it is executing a move command, as it was mentioned before only one move command a time can be executed. Next, the system was tested moving the robots from the Center for Intelligent Systems located at ITESM Campus Monterrey, Mexico. Here, the latency on the Internet affects the vision feedback, but still the time response is kept under working conditions. Besides this delay on the vision system, only other problem arose in the second test. This is related to the pan-tilt server. Specifically, this is about how the Inter-operable Object Reference (IOR) is created. In this case, the computer used as the pan-tilt server is defined into a local domain, in such way that only computers from the local domain can access the object server implementation. This problem was fixed by changing the domain definition of the computer without modifying the applications. Due to this was the first application we developed, the training time for learning CORBA functionality and implementing its functions took around three months.

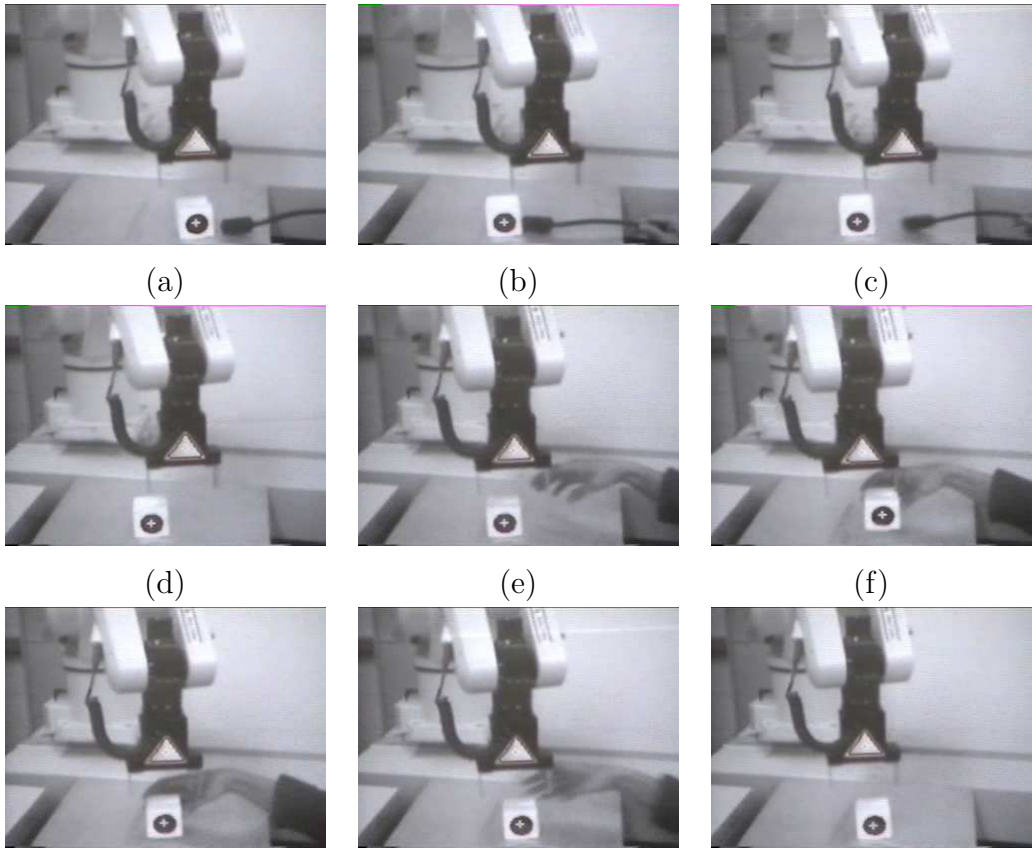
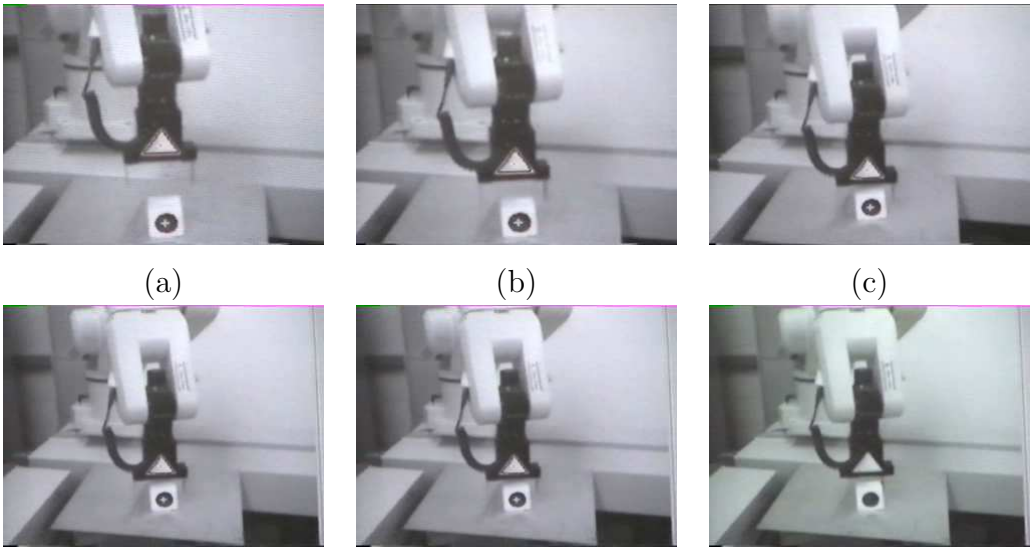


Figure 6.37: GRASP command execution with changes on the environment, Part-I. The following images shows when the artifact to move appears on the screen (a). Next, in (b) the block is moved to the left (from the user view point). In (c) the artifact is taked apart. In (d) and (e) is observed how the robot arm is correcting its path moving to the left (again from the user point of view). In (e) and (f) Dr. Song's hand is taking the object and move it to other position, specifically a little bit to the back and to the right. In (g) the robot is correcting the path. Next, in (h) Dr. Song is not taking the object any more. In (i) it can be observed a small misalignment between the two objects' centers.



(a) (b) (c)

(d) (e) (f)

Figure 6.38: GRASP command execution with changes on the environment, Part-II. In this small sequence, in (a) the robot arm is making the alignment on axis X (a). Next it is moving on axis Y (b) and (c). In (d) and (e) the robot arm is doing a small alignment on depth. Finally, when the tracking process ends the image frame return to color mode and there are not objects with borders (f) and the gripper is closed.

6.4.2 Vision Server

Creating a generic vision server for block-word problem manipulation has several challenges. First, there is a requirement to provide precise and fast response for tracking objects. Second, the service can be required from more than one client. Due to these requirements some constraints were imposed into the system to enhance its performance. As we mentioned the objects were well defined and their representations were also easy to distinguish. Even though, we had critic problem with illumination issues. Also, because of the size of the objects the statistic information was poor in the sense of most of them expose the same behavior, in other words, most of them appear as a black spot in the image. Due this we increased certain number of features to use them as a discrimination aspect. We also used other approaches to weight the parameters according to the information in the database, but the problem with this approach is that it increases with the number of objects stored. In the other hand, the interline method used for tracking is very fast to find the center point of the object but it has the drawback of using a closed-contour. In some cases, when the illumination is poor the closed-contour exceeds the maximum number of points allowed for an object.

The development time for this application was very reduced for CORBA issues, less

than one month was necessary to integrate the client-server application. The biggest time was in the tuning of the vision methodologies to manipulate the images frames. For sending information to clients a LISP format was used. This format is very practical because allows changing-length messages and also it allows different responses through the same channel. The client can easily manipulate the information with simple parsing methods to identify the type of message and to extract the required data. For tracking objects, a constraint is imposed, if the end-effector is in the tracking listing then it appears at the beginning of the list.

6.4.3 Autonomous distributed robot system

For this evolution stage the previous developments were used and integrated smoothly. There is a key piece of code which is critical for the success of the integration and this is the coordinator. But at the same time the effort carried out in this code is greatly reduced by the way in how the system was divided and how the main functions of each component were wrapped. A critical issue in any integration effort is the way different entities communicate each other. In our case we tried to maximize decoupling among components in order to reduce the blocking stages. A publisher/subscriber communication method using the Event Services results in a good decoupling mechanism which provides a good performance for a small number of components. In our testing we only have a problem with the latency which increases in the way more clients were subscribing in the channel. We already observed these phenomena when we were processing an image stream using several methodologies [Guedea *et al.*, 2002]. Fortunately this problem can be solved using commercial products which provide Notification Service and also a multi-cast communication schema. In the other hand, the coordinator must react to the information provided by the vision server to close the gap between the object positions, but the robot server only can manage one command a time. We fixed this problematic using a second object inside the robot component. In the case of CRS robot, the legacy interface provides a mechanism to interact with the robot controller using two robot objects. The first robot object is used to command the arm, while the other robot object is used to monitoring and stop current command when necessary. These objects create a resource conflict that is partially solved by using semaphores for concurrency. Other critical factor for integrating components is the number of connection points among them, so one important thing we kept through all design steps was to have a minimum but powerful and abstract number of functions. In the case of the vision server there are only six functions while the robot server has only fourteen functions and some of them very specific and easy to use. With respect to the planner, we decided not to send all operation sequence to the coordinator, this means that the coordinator receives the next operation once it has finished the previous one, and at this moment it can decide

based on the vision information, if the plan must be rebuilt or if it just execute next operation. The current design provides an enhanced dexterity for the arm manipulator and now, the distributed robotic system is capable of realizing a set or sequence of small tasks to achieve a final goal without or minimum human intervention. Furthermore, the arm manipulator is able to overcome some environmental changes while it is pursuing the goal.

.

Chapter 7

Conclusions

The integration of different components over a set of computers has been proposed using an object-oriented Client-Server approach with a Wrapper Component integration methodology. The Wrapper Component concept isolates the specific details of each component and provides the necessary abstraction level to facilitate the design phase. Furthermore, there is a matrix guide selection which help in the definition of the most recommendable connection scheme according to the interaction among the components. This architecture provides the advantage of reusable code for basic components, with the option of providing maintenance without changing the other components. For example, the Robot Server implementation can be modified to deal with other types of robots without disturbing other applications, such as the robot graphic user interface. In addition, this methodology speeds up the development process. In our project, a set of different wrapper components to solve a classical block-world problem were implemented and tested in three months once the interfaces were defined. Next, the same robot interface was used to integrate the robot with other robots and equipment such as commanding robot through speech-recognition, and commanding robot in a master-slave configuration. In the first case, the integration effort took 8 hours. Meanwhile the master-slave configuration took around one week.

7.1 Initial questions

Prior to our research we proposed several questions to be addressed:

- *How to integrate different types of robotic components?*
- *What is the structure that these components must have to work as a team?*

- *What are the tools that these systems must have to adapt to uncertain environments?*

After analyzing different works, projects and the research of other robotic teams, we concluded that ***standardization*** is a key issue to integrate heterogeneous components. We proposed to standardize the communication media between the different distributed robotic components using a generic specification such as CORBA, as well as other approaches like DCOM, and Java Beans, among others.

But CORBA is just a specification. By itself is not the solution to all kind of problems. Then we proposed a simple but powerful methodology to create "building blocks". We named this methodology *DWC (Divide, Wrap and Connect)*. Although at first glance this looks simple, there are some implications in each step that affect the overall design.

Dividing is the most easiest task of the three steps. In many real situations the different components are already divided by physical limitations, hardware, operating systems, and performance issues among others.

Wrapping is the first difficult step when designing the IDL interface for the component to be integrated. The definition of this interface is critical for the future integration of this component into a larger and more complex system. This work still has some art involved. We provide some steps to improve the definition of this interface:

- *Abstraction*, the IDL interface must be abstract enough to support not just the current equipment or system, but to support other current equipment or future changes in the equipment.
- *Monitoring*, each wrapper component must provide a mechanism to monitor its current state. This is mandatory in order to provide a reconfiguration of the client side with respect to the server side. This aspect helps to deal with the small differences among object implementations.
- *Configuration*, each wrapper component must provide a mechanism to change its current state according to the requirements of the environment.

Connecting is also a very important issue when integrating wrapper components. CORBA already solves the problem of working on different platforms, with different operating systems and languages. There are other services that can be explored to improve the design of the component. We proposed the use of a loosely-coupled communication mechanism named *publisher/subscriber* by using the Event Services. This mechanism eases the communication of massive messages over the Internet and provides a non-blocking scheme that relieves the server or client of complication regarding message

passing. Only in important issues a blocking message is used to ensure that only one command is executed at a time, such as the robot server.

The way an IDL interface is defined affects both the wrapping issue and the communication scheme, so a trade-off between these two topics is always on the table.

Once the architecture and methodology are established, we answer the following questions by integrating components that have some tools from other fields of artificial intelligence. The tools resemble some of the human activities such as decision-taking, planning, learning, recognition, classification, among others. We wrapped and connected a planning server to generate a sequence of operation (actions) based on a final goal and the current state. We also developed a vision server which provides learning, recognition and tracking objects functions. Finally we orchestrated all these components by mean of a coordinator module. We followed an incremental approach. First a remote operated robotic system with limited vision capabilities was created. Next, the vision system was enhanced and the interface for a generic planner was developed. Finally an autonomous distributed robotic system was integrated with the previous components to deal with the *block-world* problem. This integration is the proof that with including "smart" components the overall system can behave in an intelligent manner under specific circumstances.

Furthermore, we worked in other integration projects where our IDL interfaces were used to command the robot server from other robots or from a natural language interface. We did this without any change in our code.

7.2 Limitations

Although the methodology presented offer several advantages for the integration of dissimilar components, its application is restricted in the communication part to the services offered by the different implementations of CORBA, particulary Event Service. In case we need to integrate components developed by other distributing technologies such as DCOM or JavaBeans, then an intermediate element (i.e. translator) is necessary to connect these technologies.

On the other hand, the methodology is presented as an *Integration* tool to facilitate the connection between components, rather than a design tool or optimization tool for a particular problem. In other words, the methodology provides a sequence of steps to help in the definition of the component interface so the integration problem can be tackled effectively, but not for helping in how to design the implementation of the component, neither how to model a solution for a specific problem.

For example, on the vision system an abstract interface was defined to manage two types of images information and two levels of abstraction (raw images and object tracking information). But how the the images are processed (enhanced, compressed, decompressed, filtered, etc.) and how the object tracking is implemented (template matching, stereo processing, geometric models, data structures, etc.) are not defined by our methodology.

In that sense, we can argue that the components will be able to connect each other seamlessly but there is not guarantee that they will perform in an optimal way. To construct an optimal system, there are other research works that can be observed, such as [Borstel and Gordillo, 2004] for building Virtual Laboratories, [Li *et al.*, 2005] for building collaborative robotics, [Pablos, 2004] for creating optimal paths on object manipulation of robotic assemblies, and some of the works shown in Chapter 2.

7.3 Scope of Applicability

We used our methodology to integrate a set of distributed entities to conform a robotic system into a local area network. These entities were the robot system itself, a vision system, a planning system and accessories such as pan-tilt units. These elements were integrated to conform a distributed robotic system that can be managed remotely and that can perform some autonomous tasks such as learning, recognizing and grasping objects. Even though, the components can be located in different computers they were visualized as a single system, i.e as a robotic cell, with several attributes and abilities.

Accessing the different components was transparent for the user or operator. But multiple concurrent access to the same resources is not prevented by our methodology. The question here, is that if our methodology must include it or it must be considered by another methodology or architecture. Due to we remarked the loose coupled scheme as a better mechanism for integration, there is a trade-off to deal with when the distributed system must have a tight security mechanism. The applicability of our methodology will be diminished in this type of systems.

In this work, in order to integrate some components into the whole system, we use an extra CPU for almost each closed system. But, if there is no space for an extra CPU and there is not way to get into the closed system then our methodology does not help to integrate these components. This could be the case of space mission equipment where

and extra CPU increases power demand, number of failure points and checking points, an program complexity. Fortunately, some researchers are working on this matter and it is explained on the next section.

In a general sense the methodology proposed in this work can be applied to integrate equipment or isolated subsystems which can be wrapped under CORBA specification. Besides the distributed robotic system, other good example could be the subsystems found along a production line. Normally, these subsystems are connected in a local area network for monitoring but the interaction among them is almost null. An operator is the person in charge to command the different subsystems. Then the operator skills and the subsystem functionality can be modelled and abstracted to create wrapper components.

7.4 Comparative issues

In order to highlight the advantages and disadvantages of the previous proposal, we present the following comparative Tables 7.1 and 7.2. The comparison is made against a) a monolithic approach, and b) distributed but not abstract approach. Basically, in a monolithic approach, all data processing is made in a sequentially way into a single computer, which manages one robot set (mainly the robot controller and other peripheral equipment such as pan-tilt unit and/or a camera). This approach is constrained to the capability of one single machine, so including more components or increasing the data processing is not always possible without degradation of the robot performance. On the other hand, with multiple separated or distributed equipment there are more parallel processing capabilities and the only concern is the synchronization and integration among these components. CORBA services alleviate part of this problematic through the Event Service, as it was shown in previous sections. When compared with similar distributed approaches, we found that creating abstract functions is a very difficult problem, but once it is finished or implemented, it creates a solid foundation for more complex systems.

Table 7.1: Comparison between monolithic approach vs distributed approach.

Aspect	**Monolithic**	**Distributed**
Modularity	Limited to single CPU	Limited to number of CPU available
Time Response	Compromised when the number of components increase	Parallel Processing improves performance
Processing Power	Single CPU	Multiple CPU
Cost	Low	High
Communication	Low requirements	Critical factor
Maintenance	Centralized	Distributed

Table 7.2: Comparison between distributed non-abstract approach vs distributed abstract approach.

Aspect	**Distributed non-abstract**	**Distributed abstract**
Modularity	Normal	Better
Development Time	Short in small projects	Longer for big projects
Reusable Code	A single change in a component could affect the whole system	Single change in a component is limited to related modules
Maintenance	Easy for small projects	Easy for any project

7.5 Future research

Future research will focus on implementing this architecture on other computational environments such as Unix systems and Java applications, or another Real-Time Operating Systems, such as VxWorks.

Future research work will be also addressed in the following topics:

1. Add new object components as mobile robots [RWI, 1999] to enhance the cooperative work.
2. Use the Notification Service on the image server to provide a filtering service to distinguish among RGB color images, gray images and two stereo images.
3. Use techniques of sensor integration to improve the information gathered from all sensors.
4. Use the new CORBA/e specification to create embedded robotic components.

7.5.1 CORBA/e for Embedded Applications

Most of the wrapper features shown on this thesis come from the idea to use a "dedicated" CPU connected through a serial link (low speed connection) with a closed-owned system. Then for getting the wrapping benefits it is necessary to use an extra CPU with a high performance but with a bottleneck on the communication link. This CPU connects with the external world and provides the abstract interfaces of the services offered by the wrapped component. Although the "dedicated" CPU does not have any other applications competing for CPU resources, the true is that there are many background operating system tasks consuming resources. The migration of CORBA's services and the ORB from this extra CPU to the equipment's CPU represents a saving on hardware and an increment on the communication performance due to the elimination on the intermediate link. But migrating CORBA to an small embedded system requires to overcome some hard challenges such as small footprint, adding of new communication technologies and management of limited resources. Fortunately, a new CORBA specification has been released on recent date, this is CORBA/e for embedded applications [Jacob, 2006][Giddings, 2006].

.

Appendix A

CORBA specification

CORBA specification is a set of distributed systems standards promoted by Object Management Group (OMG) [OMG, 2000]. The idea behind CORBA is to allow applications to communicate one with another no matter where they are or who has designed them. The basic idea is to create an interface that can be used or understood by every application on different equipment. To achieve this goal an *Interface Definition Language* (IDL) is created. The CORBA specification follows a Client/Server approach. Usually the Server describes its services through this interface. Every application that needs to share its executable code must create an IDL file to be distributed over the network. The IDL file follows a nomenclature that is very similar to C++ or Java language. This file is processed according to the development tool (C++, Java, Python, etc.) and platform, Figure A.1 shows an example of these steps. This IDL file is very important due to in a *standardized* format describes the data structure (if any), attributes, modules and interfaces provided by the object implementation of the server side.

CORBA is based on an Object Request Broker (ORB), a mechanism through which distributed software and their clients may interact. It specifies an extensive set of bus-related services for creating and deleting objects, accessing them by name, storing them in persistent store, externalizing their states, and defining *ad-hoc* relationships between them. This module takes care of the interfaces and makes the changes needed to transmit and to marshal data, as is shown in Figure A.2. CORBA specification provides a method of creating interfaces between equipments to facilitate their communication. It is based also on an object-oriented design and implementation.

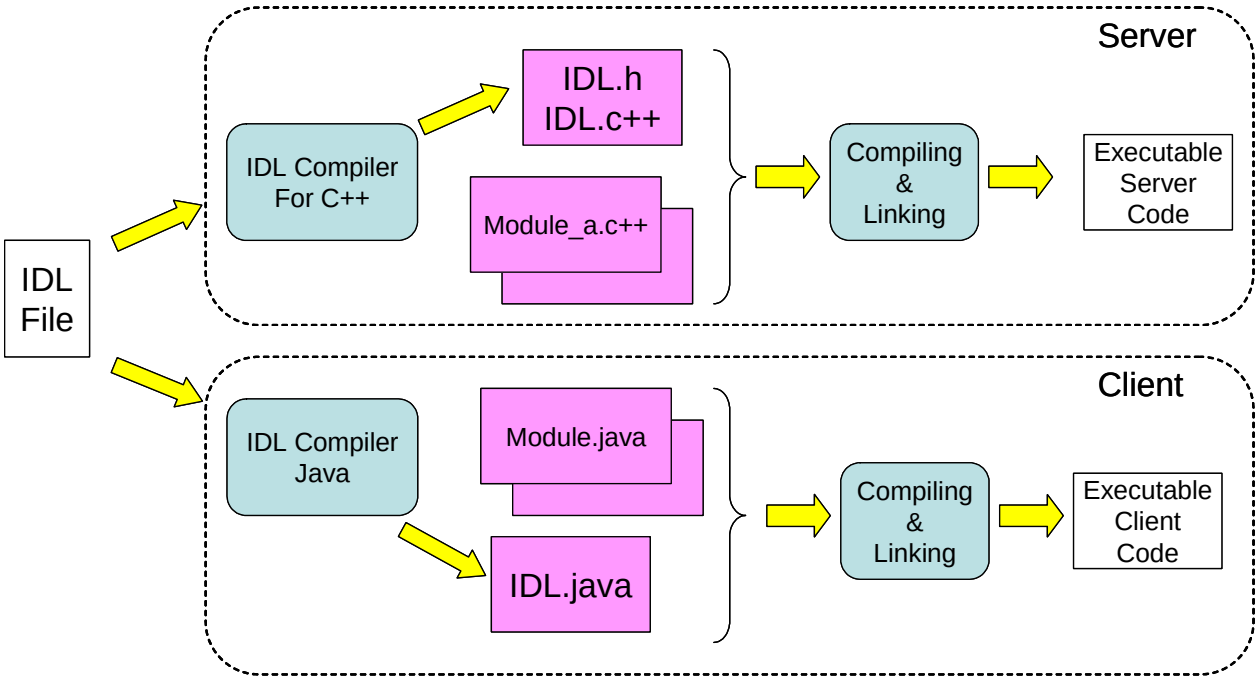


Figure A.1: IDL file processing for different development environments. The Server side is developed under C++ and the Client side is developed under Java.

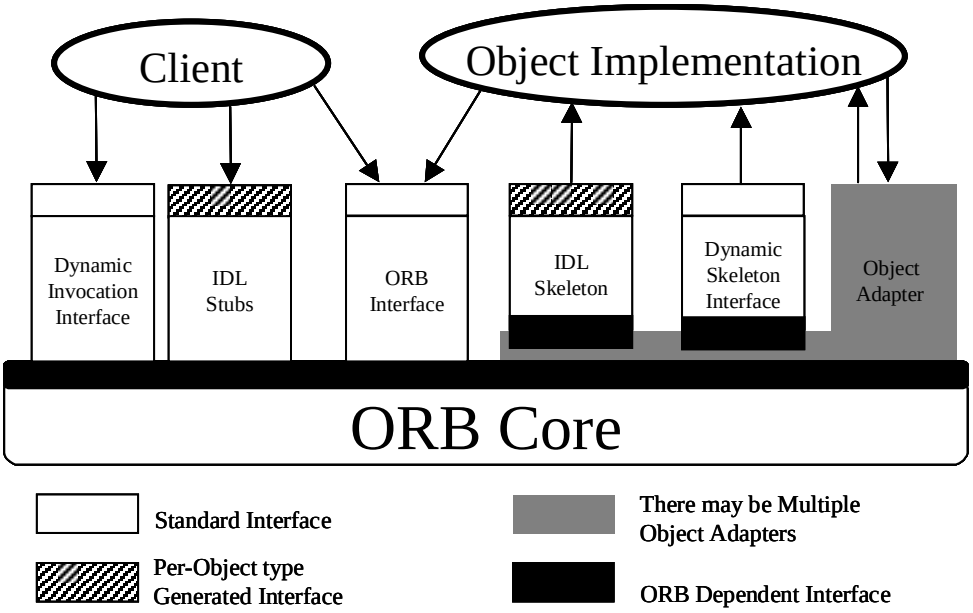


Figure A.2: Object Request Broker (ORB) interface

A.1 History

The OMG has more than 800 member companies that have been working on the CORBA standard for years. CORBA 1.1 was introduced in 1991 by the OMG and defined the *Interface Definition Language* (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0 adopted in December 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Since 1989, the OMG has been working to create standards for object-based component software within a framework of its Object Management Architecture. The key component is the Common Object Request Broker Architecture (CORBA). Since then, the world has seen a growing list of CORBA implementations come to the market. Dozens of vendors have recently announced support for the CORBA Internet Inter-ORB Protocol (IIOP), which guarantees CORBA interoperability over the Internet. Specifications of several generally useful support services now populate the Object Services segment of the architecture, and work is proceeding rapidly in specifying domain-specific technologies in many areas, including finance, health care, and telecommunications.

A.2 CORBA Architecture

The five main elements of the object management architecture, shown in Figure A.3, are:

ORB: defines the object bus and is the middleware that establishes the client/server relationships between objects. The ORB provides interoperability between applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

Object Services: define the system level object frameworks that extended the bus. The include services such as security, transaction management, and data exchange.

Common facilities: define horizontal and vertical application framework that are used directly by business objects. These deal more with the client than the server.

Domain interfaces: interfaces like common facilities but are specific to certain domain, such as manufacturing, medical, telecommunications, etc.

Application Objects: objects defined by the developer to solve the business problem. These interfaces are not standardized.

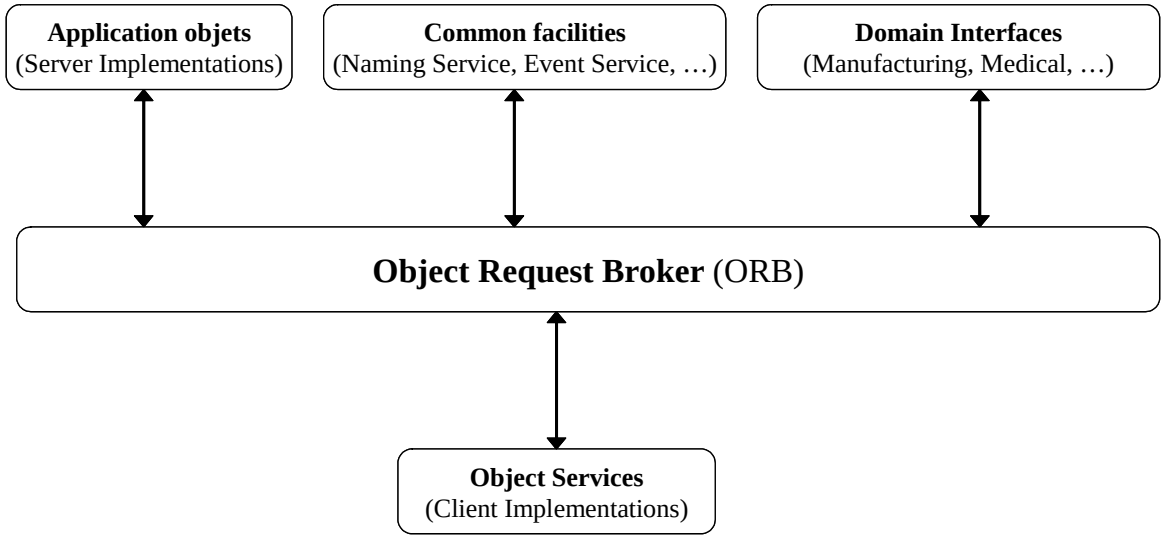


Figure A.3: Main components of OMG specification

For applications where such interfaces are not stable or static, there is an option to manage *Dynamic Interfaces* , but this option requires more programming overhead.

A.3 Interfaces and Services

In this chapter we explain how to define a generic static interface for a robot and how to use CORBA Naming Service. Naming Service is also a useful tool to look for the current object references of specific implementations. Every time a Server application starts, a new object reference for this application is created. Using Naming Service this object reference is "registered" with a specific name. So, if a client application that was "connected" with a previous server session (not longer available), tries to invoke an operation, then the call will go through an exception error because of the previous object reference is not working any more. To avoid the above mentioned problem, client application can go through the Naming Service and "resolve" the new object reference of the Server application using the corresponding name.

Our explanation begins with a very basic but generic interface definition for a Robot Server and the steps carried out on both server and client implementation in order to communicate. We used commercial ORB implementation provided by IONA: ORBACUS. Next, the development evolves to include CORBA Naming Service.

Static Interface

In order to explain the basic components and procedures involved on CORBA complaint

applications, we will start with a very simple interface. This interface is created for our Robot Server, and it has the following basic operations:

- Get_status()
- Do_action()

The first operation, Get_status(), is used to monitor the internal status of the robot. In this example, a state is returned as a string value. The default value is "NO_ACTION". The second operation, Do_action(), is the action commanded by the "Client" to the Robot Controller. Any commanded action is taken as a new state for the Robot Server, so any time the "Client" requests the status, the Robot Server returns the last commanded action or the default state if it is the first time. In real versions the status changes according to the real action of the robot.

The IDL file that defines this interface is as follows:

```
//  Robot.idl   IDL(Interface Definition Language)

interface Robot {
      void   get_status(out string status);
      void   do_action(in string action );
};
```

In this example, the interface is defined forwardly to both server and client, and it is called the *Static Interface*.

One of the main functions of the ORB is to look for objects requested of it, and to check which operations are available from each object. When one object implementation is using CORBA specification all transactions are based on an object reference. An object reference is the *unique* and *universal* identification of the object implementation. This is a standard definition shared by all operating systems, network protocols and development tools.

The ORB receives the information about Object Server Implementation and stores that information in its database. When a client application asks for that object the ORB looks into its object database and responds according to the current situation. A client application needs to invoke an operation using an object reference, and because of this, it needs a way to know this object reference.

In the following example, *reference* files were used to solve this problem with the following steps, Figure A.4:

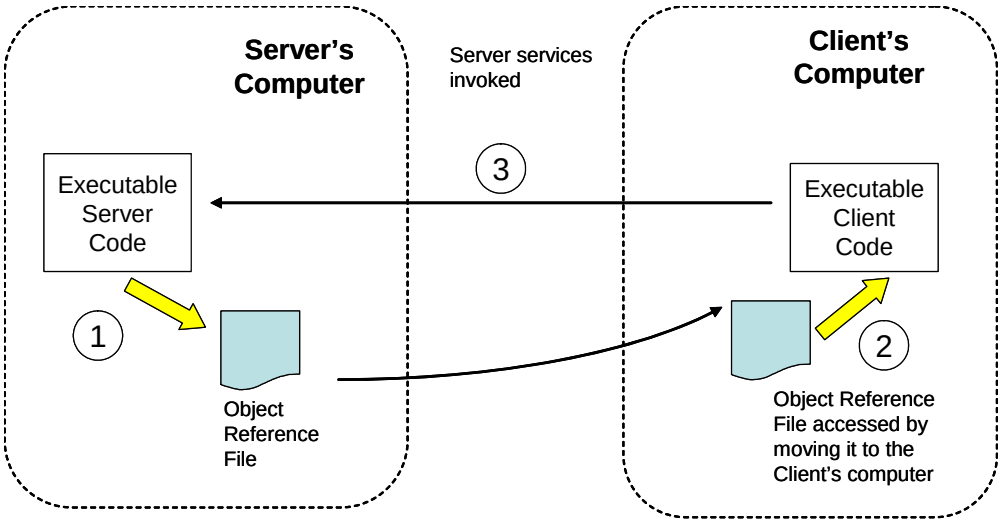


Figure A.4: Accessing an object implementation through its object reference.

1. The Server writes an object reference in a common file, in this case named "Robot.ref".
2. After this, the Client reads the file to get the reference and
3. finally invokes the operations through the ORB using that reference.

A.3.1 CORBA Naming Service

Using static interfaces we have the following problem: if the Server shuts down for any reason, i.e. power off, a fault problem or a normally operated shutdown, it has to start again and the reference for the new object implementation changes. Because of this, the last reference is not longer working. Then, Client programs, with the old reference, will not work either. They must start again and obtain the new reference. To avoid or improve this behavior, the CORBA specification offers a service called "Naming Service". With this service, Server programs "register" their implementations using names. Now Client programs only need to know two things; a) Where the naming server is and b) the name of the specific implementation needed.

With the name, Client programs "resolve" the current reference and use it with its operations. In our small example we have two operations:

- Get_status()
- Do_action()

These operations can be the same for any Robot Controller. Due to this, the Naming Service uses the concept of Contexts. A context is like an environment for several specific

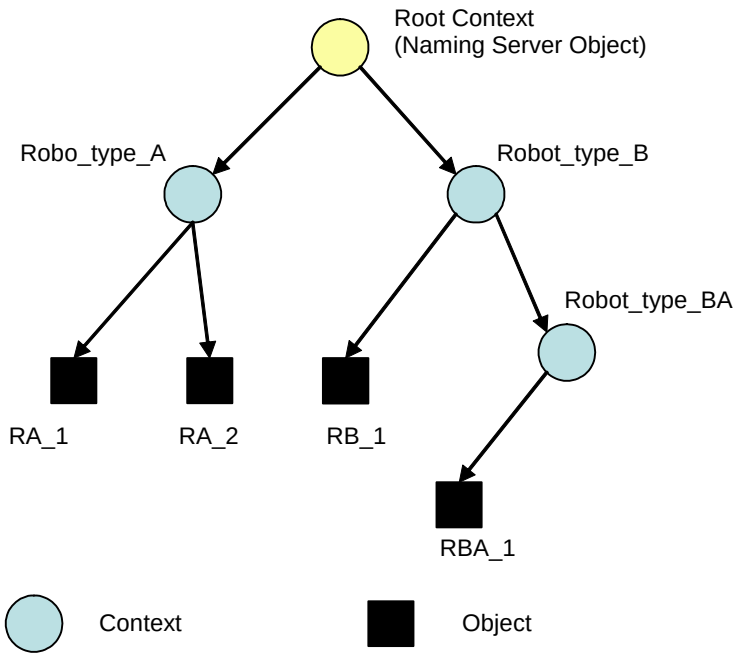


Figure A.5: Naming Service Context and Object Reference

applications. For instance, we can define the context *Robot_Type_A* to identify all the operations of robots type A. This can be shown as a hierarchical tree, where the root of the tree is the Naming Service object. Then we can create branches for each different kind of Robot and then give a special name (enumerated name for instance) for each Robot Controller implementation. See Figure A.5.

In this scheme, Client programs only look for the name of the robots, for example: Robot "RA_1". Under this configuration Server and Client programs need to know in advance the names for the object implementations, or at least the Client programs need to know the object type. The Naming Service provides functions to travel through the tree context.

Given the overview of this service, we have the following functions for each object of the class of NamingContext:

- Bind_new_context()
- bind()
- resolve()
- unbind()
- rebind()

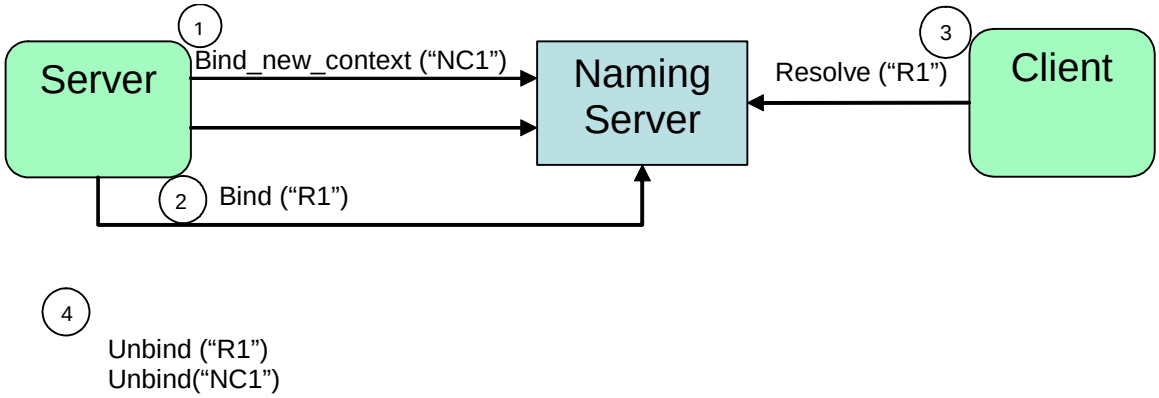


Figure A.6: Normal sequence to register, resolve and unregister the object implementation.

We also have the following classes:

- BindingList
- BindingIterator
- BindingHolder

The last three classes allow travel through the tree context. The normal process to register and to resolve the references is shown in Figure A.6. Here the first step is to register the context for each kind of robot (step 1), or for the type of robot that the Server supports. Then the Robot Implementation Controller named in this case “R1” (step 2) is registered. After this, now the Client program can ask for the Robot Implementation (step 3) or can list all the context and robot implementations registered at this time in the Naming Services (step not shown). If everything is correct, then the client can use the operations from this object.

Until now, everything works well but there are several scenarios that both, Server and Client programs must deal with:

a) When the Clients start before the Server starts or after the Server shuts down.

b) When the Server shuts down improperly and when starting again provoke an *AlreadyBound* error.

c) The method of shutting down the Server, when there are several object implementations working.

d) When the names and types for objects are mistyped or not the same as the original agreement, i.e. by changing versions.

Dealing with different scenarios

In scenario (a) the problem is in the Client side, it has to decide if the application must finish or wait for the Server application. The first decision is easier but in the second one, the waiting time must be defined and under what circumstances the Client must wait. In our example a prompt question is defined. For example the Client must keep checking or decide to leave the application.

In scenario (b) the Server has to rebind all the names of the objects only, because the context works like folders. Then, the programming steps must follow an "ask-bind" approach, or a "bind-catch-rebind" approach. The first case is to check if the context or object exists and then act in consequence. The second case is to do the bind and catch the error. The second approach is followed because there is not any operation to check if a context or object exists.

In scenario (c) part of the problem is solved using the approaches in (b) when the Server is restarting. But to avoid the problem with Client programs using the object implementations, it is necessary to wait for the completion of several threads. The Server must also unbind all the contexts and objects registered. In our programs a shutdown process is followed where the contexts and objects are deregistered (unbind operation), then one must use the shutdown() function with a thread in Java or passing the orb reference in C++. This makes the orb object returns. The exit() function is added to exit fully. If the exit() function is not used, the Server program keeps in the run() function thus doing nothing.

In scenario (d) although Server and Client programs must agree with the names, there be a case where the Server could be changed for a new version. In this situation the best thing to do is to use another kind of service: Trading Service. Under these circumstances Server and Client programs must agree at least in the kind of object to build and to use.

Although Naming Service improves the behavior of Client-Server Applications, there are several pitfalls one may encounter using this service (taken from [Henning and Vinoski, 1999]).

Nil References: The OMG Naming Service permits you to advertise a nil reference, so when you resolve a name, it is a good practice to test whether the reference returned by resolve() is nil.

Transient References: You should advertise only persistent references in the Naming Services. If you advertise transient references and your Server shuts down, the bindings created by the server will dangle and make life difficult for clients.

Unusual Names: The Naming Services specification places no restrictions on the characters that can be contained in a name component, and it even permits the empty

string as a legal value of the id and kind fields. You should avoid the use of meta-characters such as '*', '?', '/'. Orphaned Context Take care when destroying a context. You must use an inverse sequence, destroying first the leaves and then the branches of the created context tree.

Iterator Pileup: If you iterate over a naming context, make sure that you call destroy() when you are finished with the iterator.

Iterator lifetime: Although the specification does not require this, most implementations of the Naming Service are likely to use POA with the TRANSIENT policy for iterators. This means that you cannot expect iterator references to survive shutdown of the Naming Service.

Implementations limits: Many implementations of the Naming Service have restrictions on the length of a name component or the number of bindings per context.

Intervendor federation: If you federate Naming Service from different vendors, you must check that all services can store all the names you use. If some vendors places limits on the characters that may occur in a name component or on the maximum length of a component, you may encounter inter-operability problems between the implementations.

A.3.2 CORBA Event Service

Traditional CORBA requests between clients and servers are synchronous in nature. This implies that the client and the target object are tightly coupled. Although it could be desirable to have client's thread of execution blocked until the server has responded, there are other scenarios where a more asynchronous model would be better. For example, in a vision system a client could be interested to receive an image stream instead of requesting each image. An event-driven approach in this and other many cases is more feasible than implementing the client to continually poll a server or many servers for their state. In the other hand it could be the case where several clients are registered with a server to receive specific notification about changes of the server status. The server requires in this situation to have an updated list of connected clients, but this increase the overhead and the complexity on the server side. Then a more decoupled mechanism is necessary to overcome these difficulties.

The CORBA specification provides the Event Service to deal with the above situations. The basic notion of the Event Service defines a consumer as an entity (object) interested in being notified when a certain event has occurred. Conversely, it defines a supplier as the entity in which the event actually occurs. Next both entities communicate each other over an event channel. There are two general models defined for event channel:

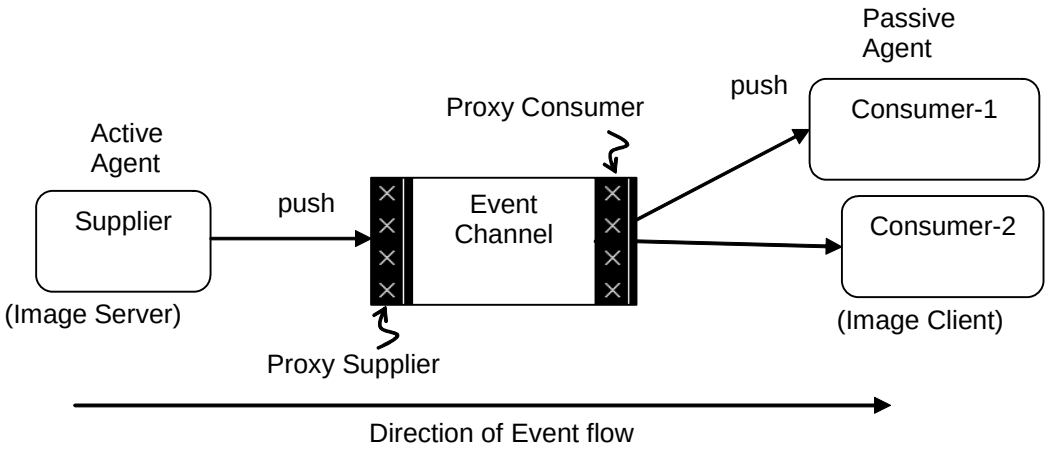


Figure A.7: Event channel using the Push Model. In this example the Image Server is the Active Agent who initiates the event delivery.

Push Model: The event is triggered in the supplier and it pushes a message through the channel, next the channel sends the message to all consumers registered in the channel. Figure A.7 depicts this model for an Image Server and Image Client.

Pull Model: The consumer requests the event channel a message which conversely pass the request to the supplier or suppliers of the channel.

The event channel has several interfaces which help suppliers to blindly send the messages through the channel, and consumers can either periodically check the event channel (pull) or request to be invoked when a specific event occurs (push). In fact many suppliers may be sending messages to many consumers, each without having explicit knowledge of the other. This scheme provides a decoupled mechanism that alleviates the burden of being waiting for a response.

For our purposes, we use the push model, and instead naming the *producer/consumer* entities we use the *publisher/subscriber* entities. The event is initiated by data publishers rather than data users, which makes data more fluent and free of much blocking. Second, it is a non-blocking communication, so even if one of the data publishers is down, subscribers can still running while the publisher is replaced by other back-up data publisher without reconfiguring the system.

A.4 CORBA in the market

In the market there are several companies that have been adopted CORBA as its option to product development, and others have been taken the standard facilities such

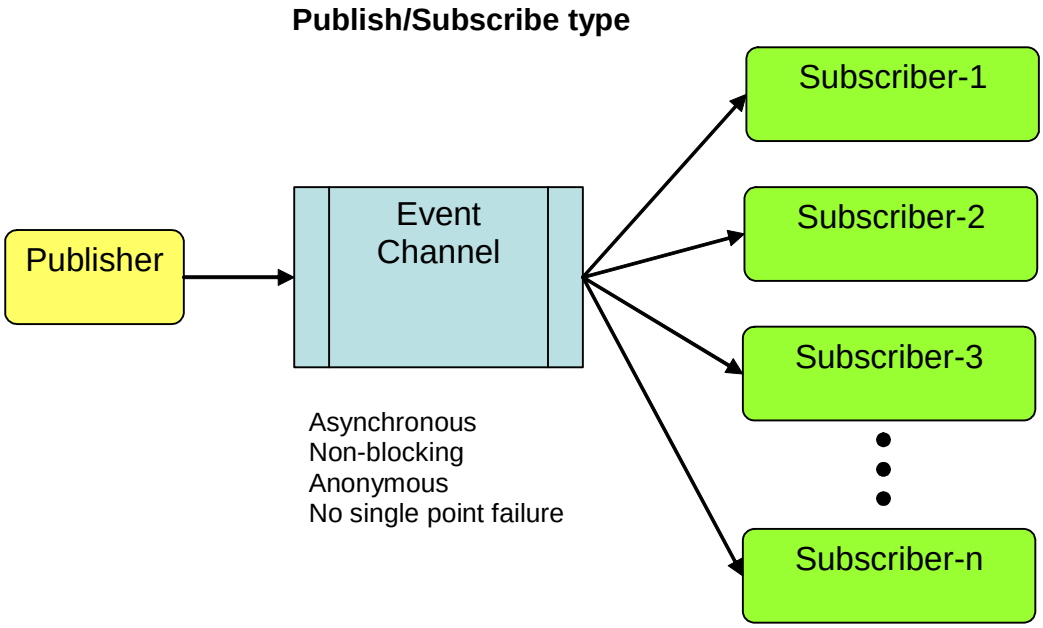


Figure A.8: Publish/Subscriber communication type is asynchronous and non-blocking communication.

as health care, financing, telecommunication, among others to create their own applications. Other companies implemented the ORB according to a specific standard version. Among them is Orbacus, Orbix and OrbixWeb from IONA (for C++ and Java applications, respectively), NEO and JOE from SunSoft, HP ORB from HP, SOM from IBM, Visibroker from Visigenic, Chorus/COOL ORB from Chorus, Dais from ICL Soft, CORBA Plus from Expersoft, OminBroker from Object-oriented Concepts, OmniORB from Olivette Research Labs, Distribute SmallTalks from ParcPlace just to mention a few ones.

CORBA is not alone in the middleware's market other technologies for object-oriented or component design and development are also present. The most important come from Microsoft's Distributed Common Object Model, JavaBeans from Sun Microsystems. Other technologies have a critical mass due its maturity in the market such as Distributed Computing Environment (DCE), or due its success to create *Internet Web applications* such as XML/WS. Table A.1 shows a comparison between them focusing in important features.

Table A.1: Feature comparison between several middleware technologies

Technology	*Maturity*	*Persistence*	*Security*	*Object-Oriented*	*API*	*Real-Time/Embedded*	*Multi-language*	*Multi-platform*	*Variety*	*Vendors*
J2EE	7	√	√	√	√	√	√	×	√	√
DCOM	8	√	√	√	√	√	×	√	×	×
DCE	14	√	×	√	×	√	×	√	√	√
XML/WS	1	×	×	×	×	×	×	√	√	√
CORBA	13	√	√	√	√	√	√	√	√	√

Appendix B

Robot Servers

The robot server component is a wrapper component that implements the generic interface definition established for this thesis work, this is shown in Figure B.1. During the development of this project, three different robots were used with the same interface. This appendix shows the details about the programming effort carried out to keep the same functionality no matter what kind of robot is being used in the project. The robots used were CRS-F3 with 6-DOF, CRS-T265 with 5-DOF and a Motoman UP6 with 6-DOF. Their main ranges and characteristics are outlined in Table B.1.

Although the three robots are jointed-arm type, the IDL interface defined as the generic interface can apply to most of the arm manipulators in the market. We will describe some of the main commands of the robot interface and how they are implemented in the different robot types.

Table B.1: Physical limits and number of axis for each robot arm manipulator.

Axis / Name (For Motoman)	CRS-F3	CRS-T265	Motoman UP6
1/S	$+/-180°$	$+/-175°$	$+/-170°$
2/L	$-135°$ to $+45°$	$0°$ to $+110°$	$-90°$ to $+155°$
3/U	$+/-135°$	$-125°$ to $+0°$	$-170°$ to $+190°$
4/R	$+/-180°$	$+/-110°$	$+/-180°$
5/B	$+/-135°$	$+/-180°$	$+/-135°$
6/T	51 turns	NA	$+/-360°$
7- Track	5000 mm	1000 mm	NA

B.1 Generic commands

The first and second commands defined in the interface are generic commands to provide more flexibility according to the capabilities offered by each robot brand. Due to its definition is a string value, a syntax in LISP or other scripting language can be used to command specific actions (*do_action()*) or to receive a message from the robot (*get_status()*). For example, the following code can be used to make a parameterized circle movement:

```
do_action("CIRCLE p1, p2, p3");
```

or the client could be interested in receiving an acknowledge that a certain file command was executed successfully:

```
get_status(FILE_CMD_STATUS);
```

where the FILE_CMD_STATUS is a string variable that is parsed to retrieve a specific information.

B.2 Retract and Extend commands

Retract and Extend commands are two wrapping commands to move the arm in similar way that a human arm. Two or more axis can be involved in this movement depending on the robot to use. Figure B.2 shows the geometric dimensions used in the CRS-F3, while Figure B.3 shows Motoman UP6. In both cases there are several configurable options to move the arm a specific distance :

a) Horizontally or keeping the vertical distance. Increment/Decrement of axis x

b) Vertically or keeping the horizontal distance. Increment/Decrement of axis y

c) Across line c or keeping the slope angle (angle α). The line between two points defined in the arm, Δc, see figures B.4 and B.5.

B.3 Movement computation for Extend/Retract commands

These movements are based in moving two axes: axes 2 and 3 for CRS-robots, or axes L and U for Motoman robots. The angles related to these axes are θ_1 and θ_1, where

```
// IDL Robot Definition
interface Robot {
// Basic function to get the status of the robot
void get_status(out string status);
// Basic function to command an action to the robot
void do_action(in string action);
////////////////////////////////////////////////////////////
// COMMANDS TO MOVE THE ARM MANIPULATOR
//

void Retract(in long distance, out boolean result );
void Extend(in short direction, in long distance, out boolean result
);
void Turn(in short direction,in short degrees, out boolean result);
void MoveH(in short direction, in long distance, out boolean result
);
void MoveV(in short direction, in long distance, out boolean result
);
void Turn_EF(in short direction,in short degrees, out boolean result
);
void Turn_Wrist(in short direction,in short degrees, out boolean
result);
void Home(out boolean result );
void Ready(out boolean result );
void Speed(in short velocity, out boolean result );
void Learn(in long var, out boolean result );
void Goto(in long var, out boolean result );
void Gripper(in long dist, out boolean result);
void Monitor(out float  x, out float  y, out float  z,
             out float rx, out float ry, out float rz,
             out float j1, out float j2, out float j3, out float j4,
             out float j5, out float j6, out float j7, out float j8);
void Finish(in short option);
};
```

Figure B.1: Basic Robot IDL interface definition to command a generic arm manipulator

Table B.2: Geometric parameter for each robot arm manipulator.

Parameter	CRS-F3	CRS-T265	Motoman UP6
a	265 mm	254 mm	570 mm
b	270 mm	254 mm	NA
m	NA	NA	130 mm
n	NA	NA	640 mm

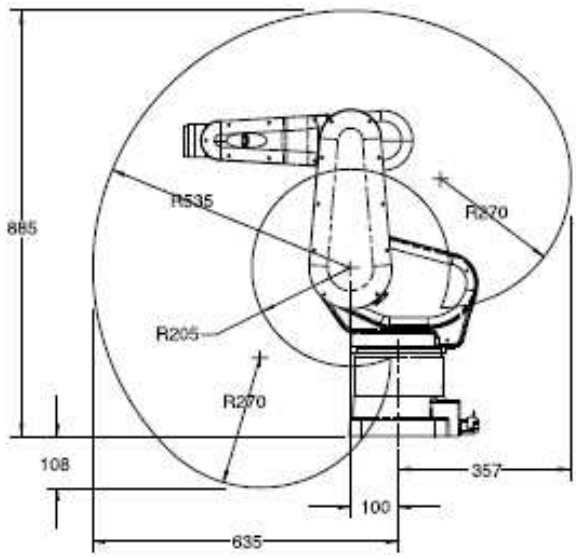


Figure B.2: CRS-F3 arm manipulator working ranges (Courtesy of Thermo Corp).

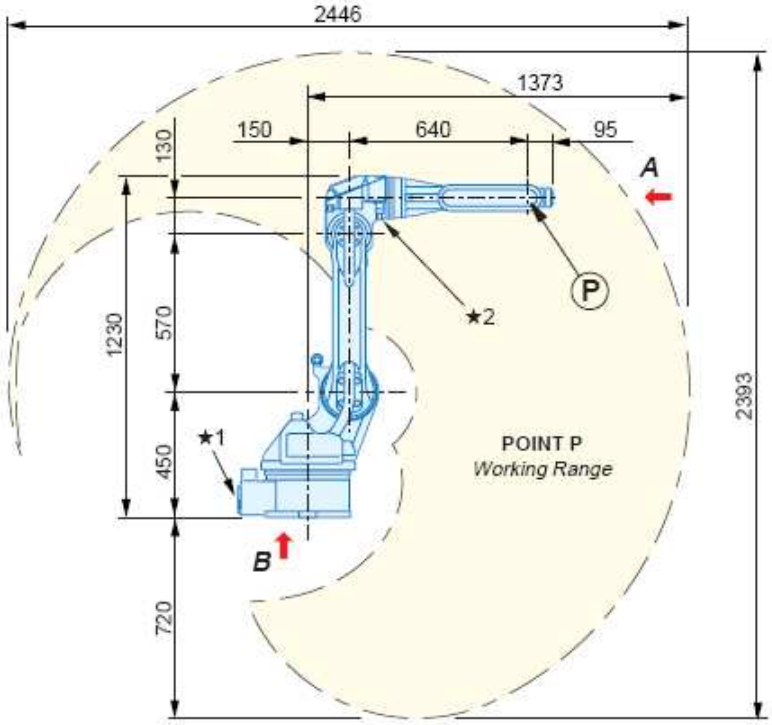


Figure B.3: Motoman UP6 working range and dimensions (Courtesy of Motoman).

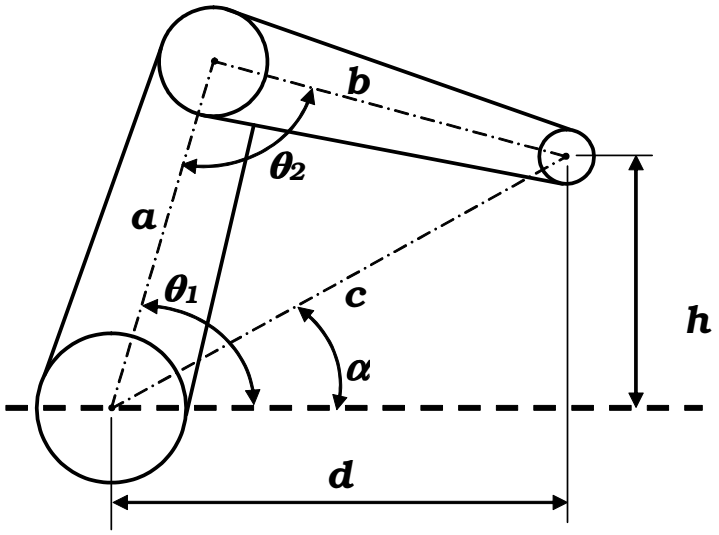


Figure B.4: CRS-F3 arm geometric parameters for Extend/Retract commands.

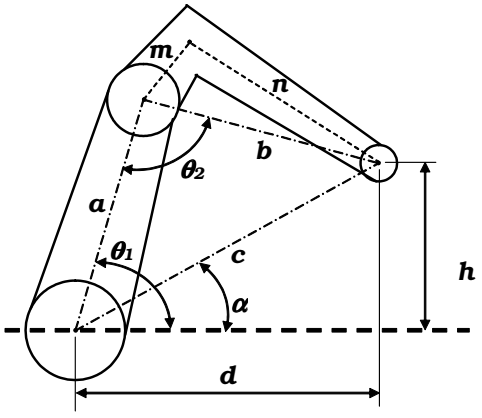


Figure B.5: Motoman UP6 arm geometric parameters for Extend/Retract commands.

Table B.3: Relationship between angles θ_1 and θ_2 and their robot arm counterparts.

Angle	CRS-F3	CRS-T265	Motoman UP6
1	90 + j2	j2	90 - L
2	90 + j2 + j3	90 + j3	90 - L + U - X

θ_1 is measured from a horizontal reference line that pass through the center of axis 2 and θ_2 is measured as the angle between links a and b. Due to the initial value of arm axes varies according to manufacturing assembly we show in Table B.3 their relationship with angles θ_1 and θ_2.

Because we will use links a and b, as our references, in Motoman case we need to compute link b and angle θ_X. These values are given by the following formulation:

Link b (Motoman case):

$$b = \sqrt{m^2 + n^2} \tag{B.1}$$

Angle θ_X (Motoman case):

$$\theta_X = cos^{-1}(\frac{m}{b}) \tag{B.2}$$

Given the above information there are several variables to compute:

Side c: here $C = \theta_2$ (from figure B.4 or figure B.5)

$$c^2 = a^2 + b^2 - 2ab\cos(\theta_C) = a^2 + b^2 - 2ab\cos(\theta_2) \tag{B.3}$$

Angle θ_B,

$$\theta_B = \cos^{-1}(\frac{b^2 - a^2 - c^2}{-2ac}) \tag{B.4}$$

Angle α,

$$\alpha = \theta_1 - \theta_B \tag{B.5}$$

Vertical distance h,

$$h = c\sin(\alpha) \tag{B.6}$$

Horizontal distance, d

$$d = c\cos(\alpha) \tag{B.7}$$

In Table B.4 there is a formulation to each option mentioned above, also figure B.6 shows the physical representation of these options.

Given the formulation, there is a back-propagation in the formulas to obtain: first, the new value *cn* and next, intermediate values until reach the new angles θ_1 and θ_1, which in turns move the specific arm axes, as shown in Table B.3.

Table B.4: Changes on formulation according to the option selected for Extend/Retract commands.

Option	Extend	Retract
Keep distance h	$\Delta d = d + \delta$	$\Delta d = d - \delta$
Keep distance d	$\Delta h = h + \delta$	$\Delta h = h - \delta$
Keep angle c	$\Delta c = c + \delta$	$\Delta c = c - \delta$

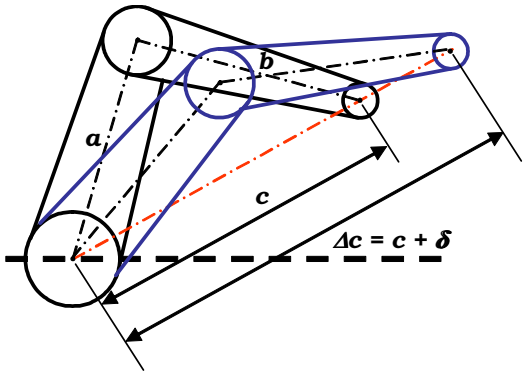


Figure B.6: Physical representation of the different options for Extend (+) command.

The Extend and Retract commands are the more complex movements to compute due the different options presented by each robot. Finally, there is a last consideration in these movements and it refers to the side-effect they have over the tool pose. CRS-F3 and Motoman UP6 have mechanically independent movement on each axis, but not in the case of CRS-T265. So, for CRS-F3 and Motoman UP6 we must compensate for the change on the previous angles that affect the tool position moving the end-effector axis. Meanwhile, in the CRS-T265 this side effect is mechanically compensated, although there is a second (non desirable) side effect in this robot. In some movements it could be the case that the angle θ_1 or θ_2 needs to move an angular distance that is far from its physical limit, but due to the mechanical arrangement the end-effector move is on its limit and the total movement can not be carried out.

B.4 Turn command

This command is very simple and it only moves the arm manipulator base clockwise or counterclockwise. There is a simple configuration in this command to establish what direction is positive or negative, and it depends only in the robot application. The argument is given in degrees and it has a resolution of cents of a degree. In all cases the axis to move is axis 1.

B.5 Turn_EF and Turn_Wrist commands

These commands move the *End-Effector* angle and *Wrist* Angle, respectively. The axis number for each command varies according to the robot used. Also the limits vary as it is shown in Table B.1. CRS-F3 has the wrist in axis 6 and it is able to make 51 turns in any directions, meanwhile CRS-T265 has the wrist on axis 5 and only can make a half-turn on each direction. Finally Motoman UP6 has the wrist on axis 6 and it is able to make a complete turn on each direction. In the case of the End-Effector, both CRS-F3 and Motoman UP6 have this element on axis 5 and they can turn 135° in both directions, meanwhile CRS-T265 has the End-Effector on axis 4 and it only can move 110° in both directions.

B.6 Linear Movements, MoveH and MoveV commands

Every robot has a base coordinates system. Although many arm manipulators makers can translate this coordinates system to the tool or to an object, we keep this coordinates system to move the robot *Tool Center Point* (TCP) along axis x and axis y. To realize this movement we can do a complex computation similar to Retract or Extend commands, but it is easier to get the current position in the coordinate system and then define a new point with the increment distance. Similar to previous commands the positive or negative direction can be configured according to the robot application.

B.7 Implementing the robot servers

In Chapter 2 we mentioned that the wrapper component hides the internal details about how a component is built. In this work we have two types of application interfaces provided by the different robot vendors. These interfaces will be "wrapped" in order to have a standard set of functions for all robots. Figure B.7 depicts this idea. The pulse shape represents the standard protocol between CORBA based application across different languages and platforms. The CORBA interface for each robot represents the developer design of an abstract interface which supports common features requested for all kinds of robots. The interaction between the API-interface provided by the robot vendors (object oriented in the case of CRS-F3 and function oriented in the case of Motoman UP6) and the CORBA interface is the most complicated job realized by the system integrator. Furthermore, in most of the cases the API interface provided by the robot vendors runs in a PC which communicates with the robot controller through

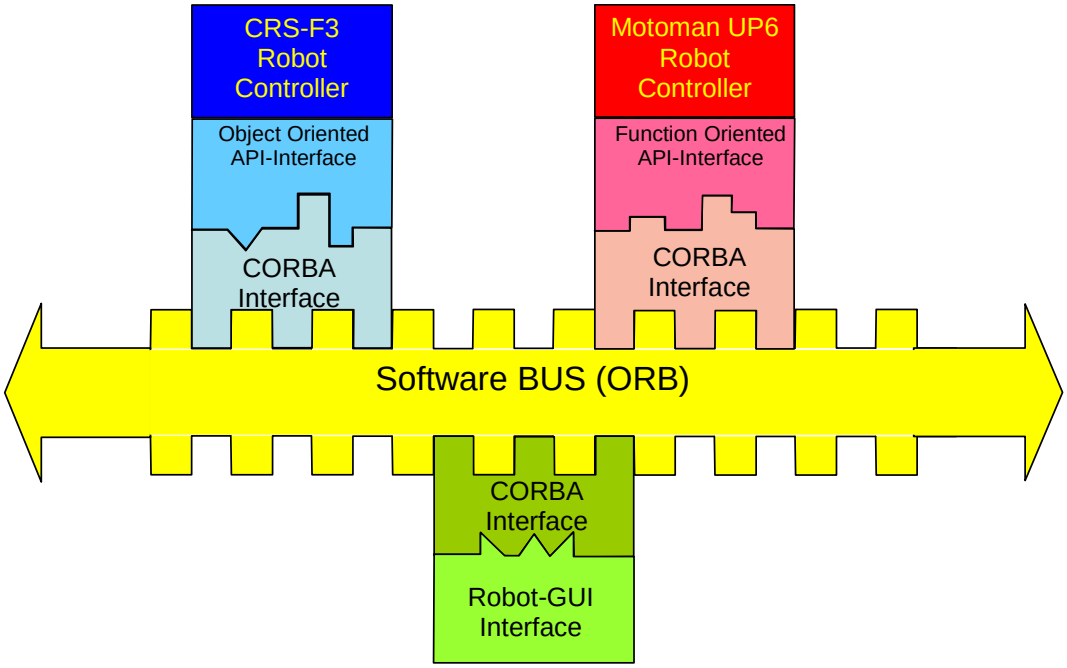


Figure B.7: Software bus concept using CORBA to integrate two different robots using the same IDL definition.

a different media, such as RS-232C (high-speed serial), Ethernet, or wireless. In the following section we will explain some details about how this task is carried out in the two robots used for this work.

B.7.1 CRS Robot arm implementation.

In the case of the CRS arm manipulators F3 and T265 they provided an object oriented interface. As a matter of fact the interface defines a set of objects through which we can reach different functionalities. Figure B.8 shows this concept.

There are several types of objects: one for managing basic arm manipulator functions (CRSRobot), while other object is defined for data file management (CRSV3File, CRSPath), operating system access (CRSRemote) and the last is used for managing locations points (CRSLocation). Table B.5 shows the number of functions provided for each type of object.

As it is shown in figure B.8, there are two user objects to manipulate the robot. One object is used for moving the robot while the other object is used to monitor the position and status of the robot controller. Both objects could access the same set of variables inside the robot controller. This creates a concurrency problem that is solved using semaphores for the shared variables. In this way the robot server can deliver an updated

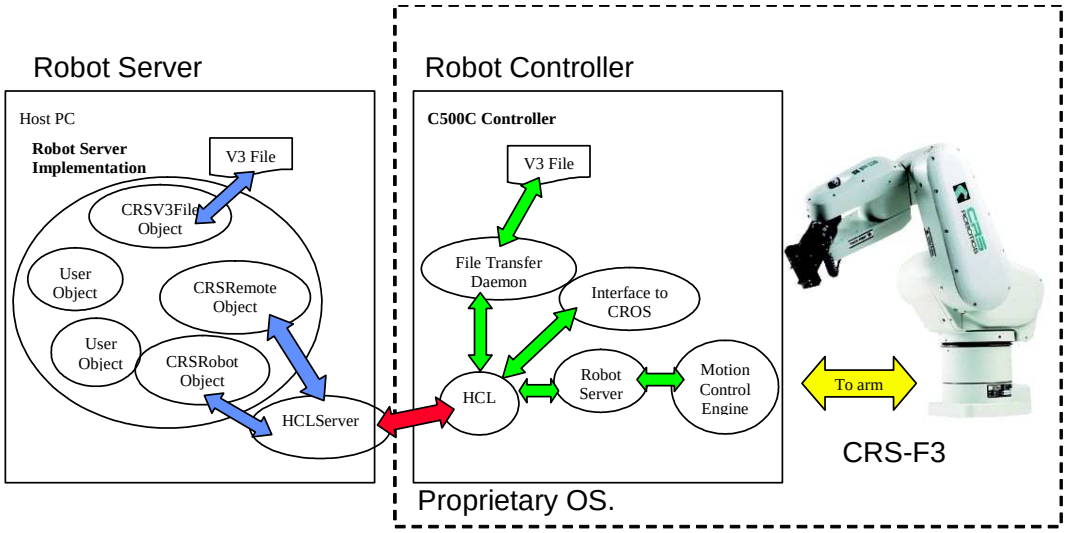


Figure B.8: CRS robot arm interface definition.

Table B.5: Number of functions for each type of object in the CRS API definition.

Object Type	Number of functions associated	Description
CRSRobot	126	For Robot Manipulation
CRSV3File	9	For Data Storage
CRSLocation	13	For Robot Location
CRSPath	5	For File Management
CRSRemote	29	For Operating System Command
Total	182	

```
////////////////////////////////////////////////////////////////////////
////  To get control of Robot Controller
////////////////////////////////////////////////////////////////////////
void RSRVMainForm::OnControl()
{
     if ( Control_Robot == FALSE )
     {
          Control_Robot = OnInitRobot();
          if ( Control_Robot == FALSE )
          {
               pButton4->EnableWindow(FALSE);
               OnAdd("Unable to get the control of the robot controller");
          }
          else
          {
               pButton4->EnableWindow(TRUE);
               OnAdd("Robot Controller accessed");
               Robot->get_RobotType(&TR);
               if ( TR == robF3 )
               {
                    OnAdd("Robot type CRS F3 with 6-DOF and one Track");
                    a = 265.0; // length in mm
                    b = 270.0; // length in mm
               }
               else if ( TR == robM1A )
               {
                    OnAdd("Robot type T265 with 5-DOF and one Track");
                    a = 254.0; // length in mm
                    b = 254.0; // length in mm
               }
               Robot_status=IDLE;
          }
     }
     else {  OnAdd("Robot Control access is ready");}
}
```

Figure B.9: Procedure to check and to take control of the CRS robot controller.

position while there is a movement in place. Due to the robot can be access through a pendant there is an initial procedure to check if the robot can be controlled, also the robot server requests to the controller the type of robot connected in order to setup specific parameters. These parameters correspond to the wrist axis and the links lengths (a and b) to use in the computation of the Retract and Extend commands. Next figure shows the code used for the initial procedure.

B.7.2 Retract command implementation in CRS robots

The following code shows some details about how the retract command is implemented for the CRS robot. In this case the retract command is defined only has a horizontal movement, i.e. the height h doesn't change.

```
////////////////////////////////////////////
// Retract Command for CRS robots
void Robot_impl::Retract(CORBA::Long distance, CORBA::Boolean_out result )
{
      char msg[200];
      double dteta1,dteta2;         // Angles of the triangle in Degrees
      double teta1,teta2;           // Angles of the triangle in Rad
      double H;                     // Height of the arm
      double a,b,c,d;               // Links and arc c to compute
      double pi,cn,dc;
      double angC,angB,angH;
      float  j2,j3,djw;
      CSingleLock RLock(&flagMove);       RLock.Lock();
      sprintf(msg,"Requesting Retract D=%d mm",distance);
      ptr->Display(msg);
      CSingleLock sLock(&(ptr->flagMutex));
      //Read Shared variables using semaphores
      sLock.Lock();
            ptr->j1=eje1; ptr->j2=eje2; ptr->j3=eje3; ptr->j4=eje4;
            ptr->j5=eje5; ptr->j6=eje6; ptr->j7=eje7; ptr->j8=eje8;
      sLock.Unlock();
      c = 0.0;          pi = 3.1415926535;      cn = 0.0;
      j2 = ptr->j2;     j3 = ptr->j3;     a  = ptr->a; b  = ptr->b;
      // Computing H and c
      // This is to make the reference 0 degrees starting at joint 1.
      if (  ptr->TR == robF3 )
      {
            dteta1 =( 90.0 + ptr->j2);
            dteta2 =( 90.0 + ptr->j2 + ptr->j3);
      }
      else
      {
            dteta1 = ptr->j2;
            dteta2 = 90.0 + ptr->j3;
      }
      // converting to radians
      teta1 =dteta1*pi/180.0;  teta2 =dteta2*pi/180.0;

      // Values to calculate: Angle C, side c, height H, distance d. angB, angH
      angC = pi - teta1 + teta2;
      c    = sqrt(a*a+b*b-2*a*b*cos(angC));
      H    = a*sin(teta1)+b*sin(teta2);
      d    = sqrt(c*c - H*H);
      angB = asin((b*sin(angC))/c);
      angH = asin(H/c);

      sprintf(msg,"Computing Angle C=%7.3f, side(c)=%9.3f,
                  Height=%9.3f, and d=%9.3f",angC*180/pi,c,H,d);
      ptr->Display(msg);
      sprintf(msg,"Extra Computing Angle B=%7.3f and Angle  H=%7.3f",
                  angB*180/pi,angH*180/pi);
      ptr->Display(msg);
```

Figure B.10: Retract command code implementation for CRS-F3, part 1.

```
      // Now, we calculate the new distance
      // delta has to be a parameter, right now is a constant of 100 mm.
      if ( abs(c - C_MIN) < 0.02 )
      {
            sprintf(msg,"Arm Full Contracted... no action taken place");
            ptr->Display(msg);
            result = FALSE;
      }
      else
      {
            cn = c - distance;
            if ( cn < C_MIN ) cn = C_MIN;
            sprintf(msg,"Retracting HORIZONTAL"); ptr->Display(msg);
            // Calculating new angC and AngB
            angC = acos((a*a+b*b-cn*cn)/(2*a*b));
            angB = asin((b*sin(angC))/cn);
            angH = asin(H/cn);
            teta1 = angB + angH;
            // Converting to degrees
            dteta1 = teta1*180/pi; dc = angC*180/pi;
            if ( ptr->TR == robF3 )
            {
                  ptr->j2 = dteta1-90; ptr->j3 = dc  - 180;
                  // To keep the end effector position
                  djw = -(ptr->j2 - j2 + ptr->j3 - j3);
            }
            else
            {
                  ptr->j3 = dc; ptr->j2 = dteta1;
            }
            if ( ptr->TR == robF3 ) ptr->j5 = ptr->j5 + djw;
            sprintf(msg,"New angle J1=%7.3f (deg) and angle J2=%7.3f(deg)”,ptr->j2,ptr->j3);
            ptr->Display(msg);
            ValidateLoc();
            ptr->Tmp_Loc = ptr->Robot->GetJointToMotor(ptr->j1,ptr->j2,
                        ptr->j3,ptr->j4,ptr->j5,ptr->j6,ptr->j7,ptr->j8);
            try
            {
                  ptr->Robot_status= MOV_RETRACT;
                  ptr->Robot->Move(ptr->Tmp_Loc);
            } catch(_com_error MyError)
            {
                  sprintf(msg,"Error!: When trying to Retract ARM");
                  ptr->Display(msg);
                  sprintf(msg,"Description:%s",(LPCTSTR)MyError.Description());
                  ptr->Display(msg);
                  CheckAbort();
            }
            result = TRUE;
      }
      RLock.Unlock();
}
```

Figure B.11: Retract command code implementation for CRS-F3, part 2.

B.7.3 Motoman UP6 arm implementation

In the case of the Motoman UP6 arm manipulator, the vendor provides a function-oriented interface. It doesn't manage the concept of objects. There is a set of global variables that the functions can fill in during their execution or the programmer must setup these variables before calling a function. Due to the robot controller can manage several robots at the same time, it is necessary to get a handler for each robot. Depending on which port the robot is connected it has a specific communication handler ID.

A joint movement function call using pulses has the following format:

```
BscPMovj ( short nCid   // Communication handler ID
           double speed, // Move speed (mm/s or /s)
           short toolno, // Tool number
           double *pos   // Target position storage pointer
           );
```

The robot could use different types of grippers or tools and each one has a number assigned. For normal gripper the tool number is zero (0). For moving the robot, it is necessary to specify a target position. This position is defined using pulses, so the implementation code must convert pulses to degrees or radians in order to perform any geometric computation. Each axis has an index position inside an array of double values.

```
//////////////////////////////////////////////////////////////////////
// Retract
void Robot_impl::Retract(CORBA::Long distance,CORBA::Boolean_out result )
{
   char msg[200], dir[20], data[10];
   unsigned short * spRconf;
   short sRobotPos2;
   CString csConv;
   spRconf=new unsigned short;
   dpPosData=(double *) new double[11];
   long dist;
   double theta_L, theta_U;          // Angles of the triangle in radians
   double dtheta_L, dtheta_U;        // Angles of the triangle in Degrees
   double dtheta_LN, dtheta_UN;      // Angles of the triangle in Degrees
   double dtheta1, dtheta2;          // Angles of the triangle in Degrees
   double theta1, theta2;            // Angles of the triangle in Radians
   double pulsos_l, pulsos_L;        // Pulses for axis L
   double pulsos_u, pulsos_U;        // Pulses for axis U
   double djw, theta_x, dtheta_x;
   double angC, angB, angH;          // Angle of the triangle
   double H;                         // Height of the arm
   double a,b,c,d;                   // Links a and b, and Arc C to compute
   double m, n;                      // Links m and n
   double cn, dc, pi;
   double cte1 = 360.0/484000.0;     // For convesion of Degree vs. Pulses
   double cte2 = 360.0/484000.0;
   double cte3 = 360.0/283600.0;     // Variables of BscMovj
   double Speed = speedvar;
   char cFrName[100] = "BASE";       // Coordinate Frame
   unsigned short RConf=0;
   unsigned short ToolNo = 00;       // Short stores the tool number
   double * dpTargetP;               // Target position storage pointer
   short sRobotPos3;                 // Return value from BscMovj
   // Starting CODE
   sprintf(msg,"Requesting to Retract in one amount of %ld",distance);
   ptr->Display(msg);
   c = 0.0; pi = 3.1415926535; cn = 0.0;

   // Working with JOINT 1 //
   sRobotPos2 = BscIsLoc(sComHandle, 1, spRconf, dpPosData);
   pulsos_l = dpPosData[1]; pulsos_u = dpPosData[2];     // Get pulse counting
   dtheta_L = pulsos_l*cte1;  dtheta_U = pulsos_u*cte2;  // converting to degrees
   theta_L = dtheta_L * pi / 180.0;                    // Converting to radians
   theta_U = dtheta_U * pi / 180.0;
   // Links a, b and distance n        and Computing value of b
   a = 570.0; m = 127.0; n = 640.0;    b = sqrt(m*m+n*n);
   // Computing value of angle X
   theta_x = atan2(m,n);
   dtheta_x = theta_x * 180 / pi;
   // Values of angles theta1 y theta2
   dtheta1 = 90.0 - dtheta_L;
   dtheta2 = - dtheta_L + dtheta_U + dtheta_x;
   // converting to radians
   theta1 =dtheta1*pi/180.0;
   theta2 =dtheta2*pi/180.0;
```

Figure B.12: Retract command code implementation for Motoman UP6, part 1.

```
// Values to calculate:
   // Angle C, side c, height H, distance d. angB, angH
   angC = pi/2 + theta_x + theta_U;
   c = sqrt(a*a+b*b-2*a*b*cos(angC));
   H = a*sin(theta1)+b*sin(theta2);
   d = sqrt(c*c - H*H);
   angB = acos((b*b-a*a-c*c)/(-2.0*a*c));
   angH = theta1-angB;
   // Validation
   if ( abs(c-a-b) < 0.02 )
   {
         sprintf(msg,"Arm full extended...no action taken place");
         AfxMessageBox (msg);
   }
   else
   {
         // Now, we calculate the new distance
         cn = c - dist;
         if ( cn > (a+b) )
         {
               sprintf(msg,"Distance is out of range making a full extended");
               AfxMessageBox (msg);
               cn=(a+b);
         }
         sprintf(msg,"Extending AHEAD"); AfxMessageBox (msg);
         angC = acos((a*a+b*b-cn*cn)/(2*a*b));
         angB = acos((a*a+cn*cn-b*b)/(2*a*cn));
         theta1 = angB + angH;
         // Converting to degrees
         dtheta1 = theta1*180/pi;
         dc = angC*180/pi;
         //New values of axis in grades
         dtheta_LN = 90 - dtheta1;
         dtheta_UN = dc-dtheta_x-90;
         //To keep End-Effector Position
         djw = -(dtheta_LN - dtheta_L) + (dtheta_UN - dtheta_U);
         // Conversion to pulses
         theta_L = dtheta_LN*pi/180.0; theta_U = dtheta_UN*pi/180.0;
         pulsos_l = dtheta_LN /cte1;  pulsos_u = dtheta_UN /cte2;
         djw = djw/cte3;
         dpPosData[1]=pulsos_l; dpPosData[2]=pulsos_u; dpPosData[4]=dpPosData[4]-djw;
         // Retracting to new position //
         sRobotPos3 = BscPMovj (sComHandle, Speed, ToolNo, dpPosData);
   }
   result = TRUE;
}
```

Figure B.13: Retract command code implementation for Motoman UP6, part 2.

Appendix C

Vision Server

The Vision server module is a wrapper component that implements the generic interface definition established for this thesis work, this is shown in Figure C.1. From a general point of view this component wraps two abstraction levels:

a) Data image capture/transmission wrapping level, and

b) Image Object information wrapping level

In order to describe how these services are wrapped, we briefly describe the fundamental stages on image processing. Next, the following sections will describe the specific details about how the parameters and knowledge was configured to overcome the problem domain of a distributed robotic application.

```
// IDL Vision Server Definition
interface  ImageServer
{
            void Learn(in short x, in short y, in string name);
            void Find(in string name);
            void Track(in string name);
            void EndTrack();
            void GetObj();
};
```

Figure C.1: Basic Vision IDL interface definition to command a generic vision server

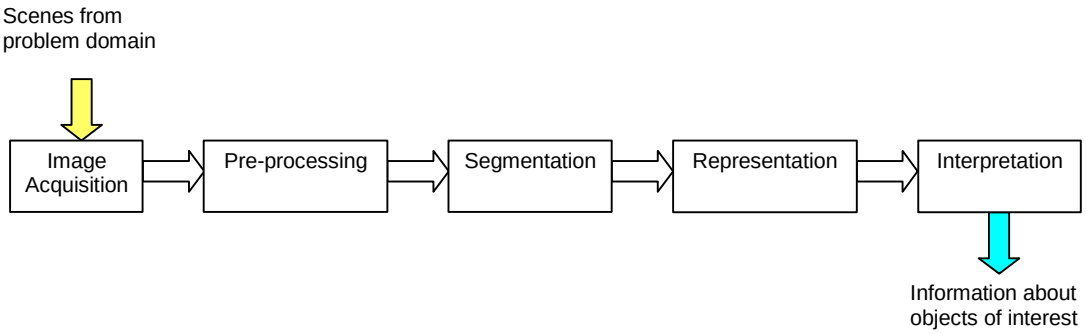


Figure C.2: Fundamental stages on image processing.

C.1 Fundamental stages for image processing

In order to provide the aforementioned services, scenes from real word must pass by several stages before useful information is extracted from them. These stages involve hardware and software issues that limit the scope and performance of the vision system, next Figure C.2 shows these steps.

C.1.1 Image Acquisition

Three elements are necessary to acquire digital images of video. The first one is a physical device that is sensitive to the visible band of the electromagnetic spectrum and that produces an analog electrical signal proportional to the level of energy perceived. The second element is a cable by which the electric signal is transmitted, and the third is a digitizer, responsible for converting the electric signal from the physical device into a digital form. Nowadays, the analog signal is already converted inside the physical device and a bit stream (digital format) is sent through the cable following a standard protocol (IEEE 1394). This improves the quality of the image but reduces the cable distance available. The image acquired has a resolution established by the number of pixels provided by the camera. In our case we used a stereo camera from Videre Design with a resolution of 320H x 240V pixels, and color capability. The model camera is STH-DCAM, and it has a frame rate of 15 frame per second (fps) using both cameras in color mode. Due this factor we choose a 10 fps rate which is acceptable for our purposes.

C.1.2 Image Pre-processing

Once the image frame is captured, the following step is the image pre-processing. This stage name is given to the low level abstraction operations in the images that do not

increase the quantity of information but suppress information that is not prominent for the main analysis objectives of a given case. The main function is to improve the image quality so there is a bigger chance to increase the opportunity of success of the following processes. Typically, the techniques involve contrast enhancement and noise removal.

C.1.3 Segmentation

The third step is the segmentation that subdivides an image in its parts or constituent objects. The level of this subdivision depends on the problem to resolve, this is, the segmentation should be stopped when the objects of interest for an application have been isolated. In general, automatic segmentation is a difficult task. Segmentation algorithms have three common forms: "Methods based on edges", "Techniques based on regions" and, "Techniques based on thresholds". Methods based on edges are centered in the detection of contours. They delimit the edge of an object and segment the pixels inside the contour as belonging to that object. Their weakness consists in connecting separated or incomplete contours, which make them susceptible to failures. Techniques based on regions, usually operate in the following form: the image is divided into regions grouped by neighbor pixels with similar levels of intensity. The adjacent regions are united under certain criterion that involves the homogeneity and sharpness of the borders of the region. A very strict criterion provoke fragmentation, a little strict criterion causes unwanted unions. Techniques based on thresholds segment the image pixel by pixel, that is to say, they do not take into consideration the value of the neighboring pixels. If the value of a pixel falls inside of the rank specified for an object the pixel belongs to the segment. They are effective when the objects and the background of the image have ranks of different values and a marked contrast exists among them. In this technique the borders of blurry regions can cause problems.

The selection of a segmentation technique is determined by the particular characteristics of the problem to resolve. The outputs of this phase are the values of the pixels that form the border of a region or the region itself. In this thesis work the objective of the segmentation is to extract the characteristics necessary of an object in movement in order to separate it from other parts of the image.

C.1.4 Representation and codification

The knowledge about the problem domain is usually codified in a knowledge database. This knowledge can be as simple as to detail the regions of an image where is known that information of interest is located, limiting in this way the search scope that should be carried out to find such information. In the other hand, the knowledge database can

be very complex, such as an interrelated list of all the possible defects in a material inspection problem. The knowledge database guides the operation of each processing module and guides the interaction and communication among them. In our case the object obtained from the previous stage are identified by a set of parameters as we will shown in the following sections.

C.1.5 Acknowledge and Interpretation

The computation in a high level includes the recognition and interpretation processes. These two processes have a great similarity with intelligent knowledge. Most of the techniques employed by the processes of low and intermediate level utilize an assembly of well defined formulations. Nevertheless, the recognition and interpretation of the information provided by those levels, become much less precise and more speculative. This relative lack of comprehension is translated into a formulation with constraints and idealizations whose main purpose is to reduce the complexity of the tasks, until achieving a reasonable level. The final product is a system with operating capacities extremely specialized. In our case these capacities correspond to the functions defined in the Vision IDL interface.

C.2 Data image capture/transmission level

This is a basic service provided by the vision server in which the image stream captured is compressed and send it through a specific channel. The channel is created using the event service defined by CORBA specification. Figure C.3 shows the steps carried out to provide this basic functionality.

Although there are several ways to transmit the image we decided to use a free length format using a string data type from CORBA specification. The only inconvenience with this alternative is that every zero value inside the stream must be transformed to non-zero value in order to avoid undesired trim of the image. Depending on the image data this transformation could generate an overhead in data and processing time around a 5% of the data size to send. The reverse procedure must be executed to recover the original image. Due to we are using a JPEG format some data will be lost in the compression procedure. The image is 320W x 240H in size for an original size of 225 KB (three bytes are used for each pixel to manage RGB color), with the compression procedure the image data is around 20 KB. This gives an 11:1 ratio compression.

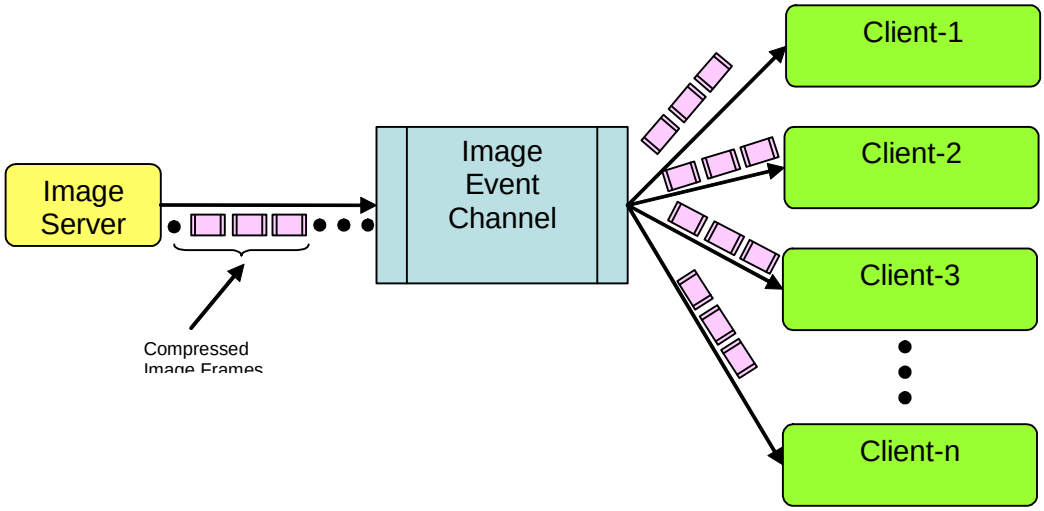


Figure C.3: Image Transmission using an event channel.

C.3 Image Object information wrapping level

To enhance the Distributed robotic system, the vision component is integrated to provide the following functionalities:

1. Image feedback to human user
2. learn specific objects defined by the user
3. find and track selected objects
4. send object position feedback to the robot brain

The image feedback is provided by the previous wrapping level and the other functionalities require more image processing. Due to these functionalities are required on a real-time base some constraints are imposed in order to improve the response time of the vision system.

C.3.1 Constraints

The objects to learn and to find must be solid objects (geometric figures), not overlapped, and only their central point location is required. This location is referenced at pixel level $\mathrm{p}(x, y)$. Due we are using a stereo-camera there is more information required, this is the distance of the central point of the object to the center point of the camera base line.

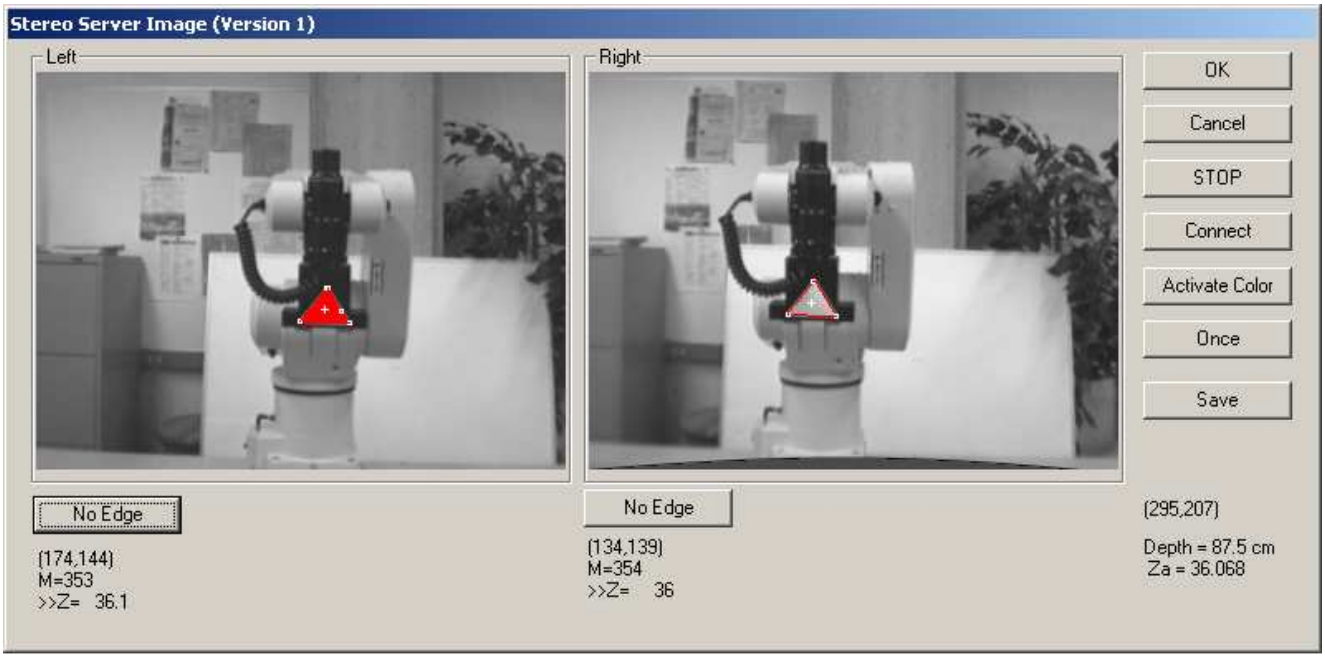


Figure C.4: Stereo images used for the learning process. For calibration purposes the left image is used as a reference while the right image is adjusted, this is observed by the curvature on the bottom.

C.3.2 Learning an object in an image stream

The vision server provides to the user with the ability to decide which object to learn. Using the image feedback the user can select an object (normally defined as an area) using the mouse pointer. The point p(x, y) selected must be inside the object area. The area selected is filled in with a specific color (red) to feedback the user about what area is understood by the learning method. This step is repeated for 10 frames and the computation for each frame is averaged. At the end, the area is outlined following its contour and several parameters are defined for this object. In figure C.4 there are two images showing these steps. Left image shows the filling process and right image shows the contour line only. Figure C.5 shows the dialog window for the learning parameters and Table C.1 shows the specific parameters calculated for each object. Mainly we defined several Hu moments and some features. The features are defined as corners and they are marked as small circles in the contour line (figure C.4). Also, the center point of the object is marked as a cross. If the final data is agreed with the user appreciation, then a name provided by the user is assigned and the object data is stored.

Each time an object is learned, a square image area containing the object is saved as a template. The template is saved as a bitmap file where the filename is composed of the object name and followed by its sample number. Figure C.6 depicts this idea.

Table C.1: Main parameters defined to identify a solid object.

Parameter	Description
Name	Object name provided by the user
Area	Average area taken from 10 image frames
Perim	Average perimeter taken from 10 image frames
NF	Number of features associated with the object
M0	Hu moment 0
M1	Hu moment 1
M2	Hu moment 2
Num	Number of samples taken for this object name

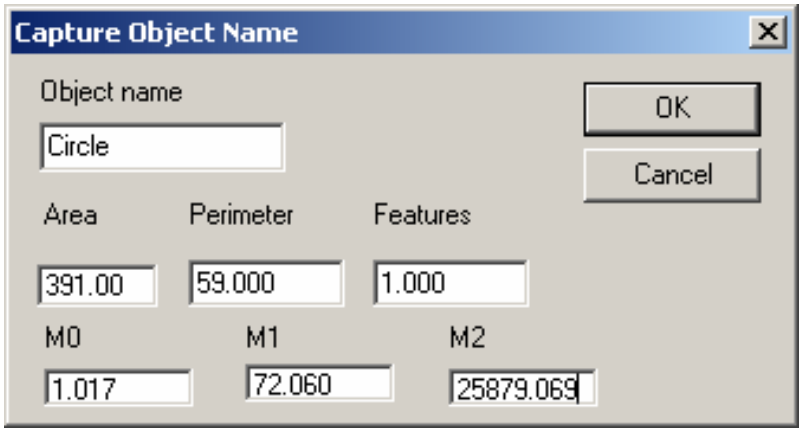


Figure C.5: Object parameter obtained from the learning process. In this case, these parameters correspond to the geometric figure of a circle.

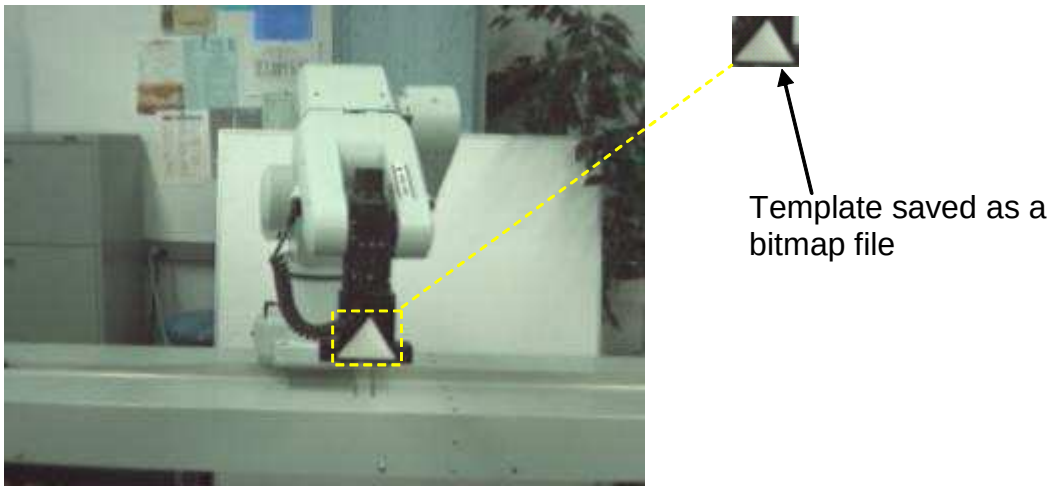


Figure C.6: Object template selected to store it as a bitmap file.

C.3.3 Finding an object in an image stream

Finding an object is a process where the previous stored information of the object is used to make a point-based segmentation. The segmentation is carried out by a template match algorithm. This algorithm creates a dynamic list where possible central point candidates are selected as possible object match. Next, due to neighbor points represent the same object, there is a filter process to select separated points. The second dynamic list is processed to evaluate the parameter of each object. In each comparison the object with the minimum Euclidian distance from the original object's parameter is selected. Also, there is a defined error threshold to select the last object as a good or bad matching. Figure C.7 shows the different stages of the Finding process.

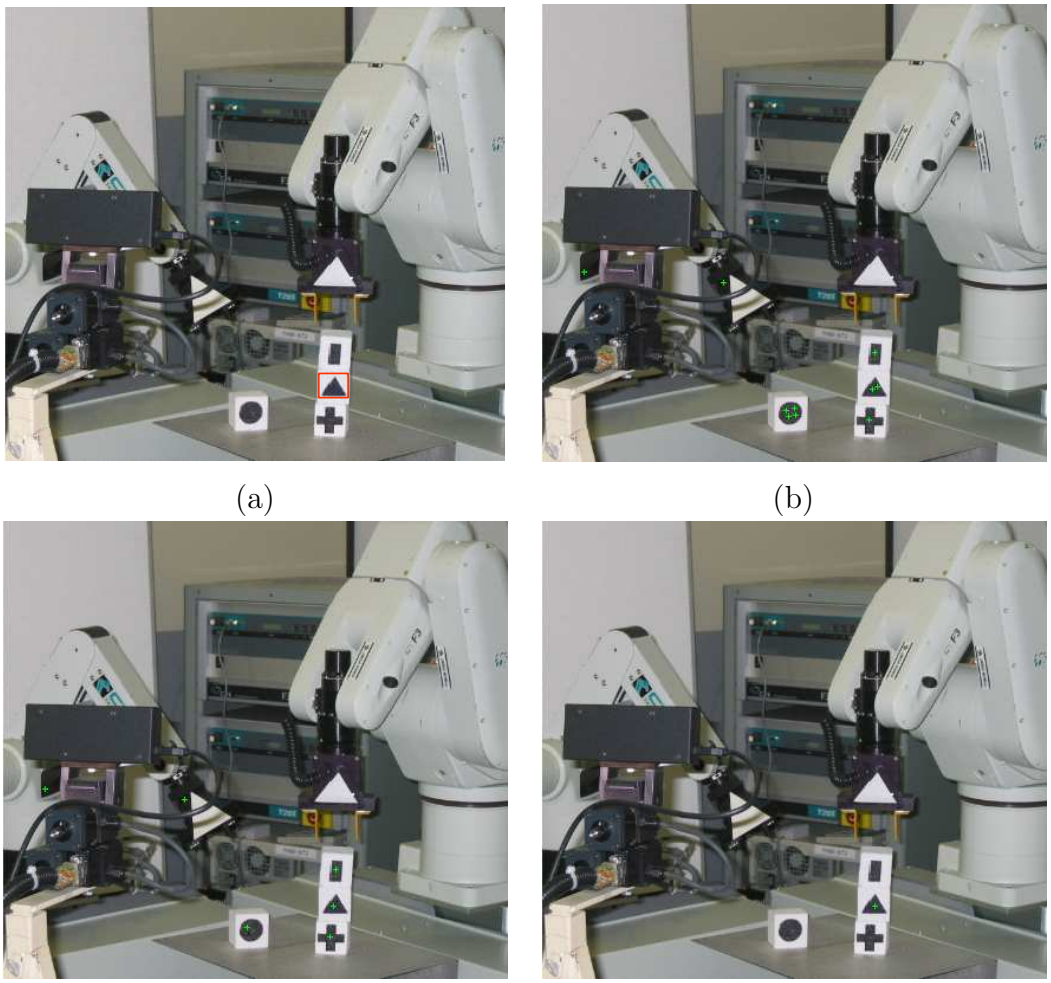
C.3.4 Tracking objects in an image stream

Finding an object is a very expensive algorithm but once the object is located it is easy to track it. For object tracking we are using the interline method [Guedea *et al.*, 2003b] which is a very fast process based on the object contour. With this method the central point of the object and its pose can be computed in few steps. The computation cycle is restricted to the contour size; that for our problem it is around 200 points at most for the geometric figures of the block-world problem. To detect the contour the algorithm starts in the previous central point of the object. The only drawback of this method is that it is based on a closed contour, which some times (very often) is affected by the environment illumination or object overlapping. Smooth movements of regular size objects can be tracked without problem, but irregular movements such as drastic change on the direction affect the tracking process. Once the vision server is tracking any object it gets into a conditional loop to track a list of objects. This is a dynamic list which can be expanded to track multiple objects. The processing time is increased proportionally to the number of tracking objects. Figure C.8 shows different stages of the tracking process.

C.4 General operation of the Vision Server

The above processes can be requested to the server in any order; the client could request to learn one object, then it can request to start its tracking or just find its current position. To deal with these requests the vision server has the following operation modes:

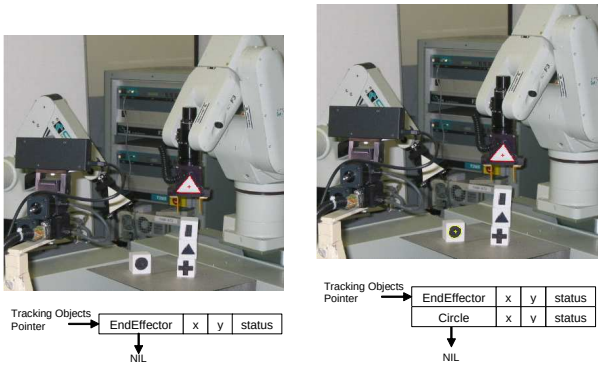
IDLE: No image is captured and no image stream is sent.



(a) (b)

(c) (d)

Figure C.7: Steps carried out to find a specific object using a mixture of algorithms. In (a) the triangle object is selected as the object to find using template matching. Due the small size of the template several options are available as is shown in (b). After a distance filter is applied only a few objects will be discriminated in (c). Finally the triangle is selected using other object features as area, perimeter or the number of corners, among others.



(a) (b)

Figure C.8: Tracking object process. The vision system has a list of objects to track where the first object is by default the End-Effector if it must be tracked. Then a chaining list is used to keep information about the last central point registered for each object. In each subsequent frame the object is located using the interline method.

CAPTURING: The vision system starts and keeps capturing images at predefined rate (10 fps).

CONNECTED The vision system sends compressed images through a specific channel. (Image Channel).

LEARNING: The vision system learns a specific object indicated by the user through the IDL interface or from the console.

FINDING: The vision system is looking for a specific object, the name of the object is provided by the user. Once the finding process is ended the vision system responds with OK or FAIL, depending if the object is localized or not. The last position of the object is also reported.

TRACKING: The vision system tracks a list of objects and reports their position in pixel dimensions, p(x, y), and their relative distance to the center of the camera. The process is started with a single command and it can be issued for different objects.

Some of the previous states could coexist but for safety operation the server cannot be at learning mode and tracking mode at the same time. The tracking process can manipulate several objects but there is only a single command to stop the tracking of all of them, there is not command to stop tracking a specific object. A graphical representation of the communication channels is shown in Figure C.9.

Finally, there is a single command which its function is to get the list of objects learned by the vision system. The response is a list of objects that have been learned by the

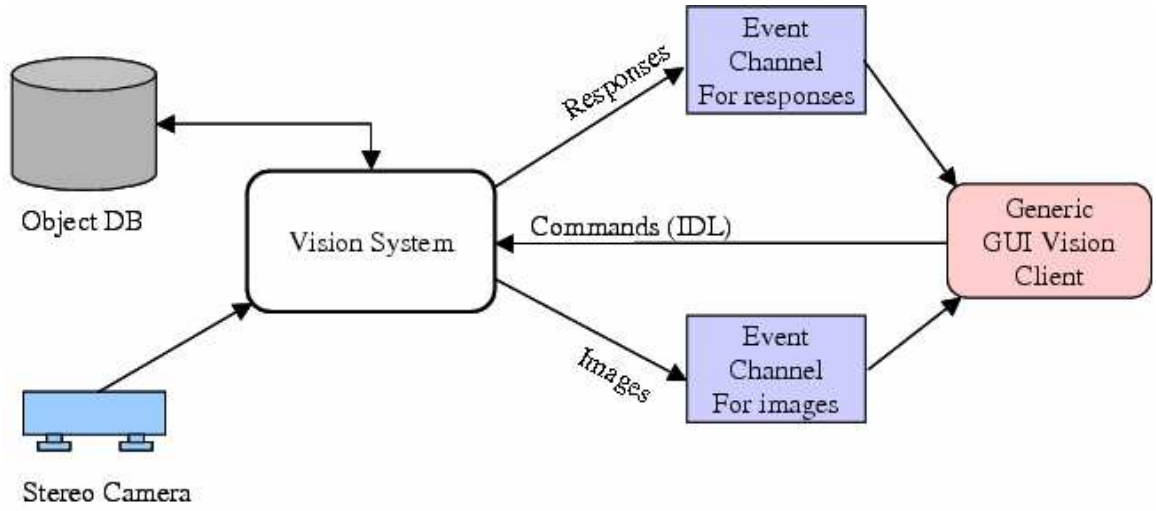


Figure C.9: Vision Server communication scheme. Due to the intense and heavy traffic of information, two different communication channels are defined for the vision system. One channel is dedicated to transmit compressed image stream and the other channel is used for object information. The vision system keeps a database of learned objects.

vision system which are stored in a database. This database could be shared with other vision systems.

For each command of the IDL interface there is an asynchronous reply. The reply could be a single response or a periodic response. Table C.2 shows the responses for each case; all responses follow a LISP format.

The vision server can be accessed through its IDL interface; one or more clients can request its services concurrently. In this thesis work the vision server is used without any access control, due to several clients could request different objects. The channel scheme helps to provide the services to many clients. Then, for future work it is necessary to have a minimum control scheme to avoid mutually exclusive operations, i.e. one client requesting to learn one object while other client is requesting to track another object.

Table C.2: Response Messages transmitted for each command of the vision server

Command	Response
Learn(x, y, name)	Single response (LEARN “name” OK) (LEARN “name” FAIL)
Find(name)	Single response (FIND “name” OK (x , y)) (FIND “name” FAIL)
Track(name)	Periodic response (TRACK number_of_objects (“object name 1” OK (x, y, d)) . . . (“object name n” FAIL))
EndTrack()	Stop all messages of the tracking process
GetObj()	(GET number_of_objects (“name 1”, . . . , “name n”))

Appendix D

Planning Server

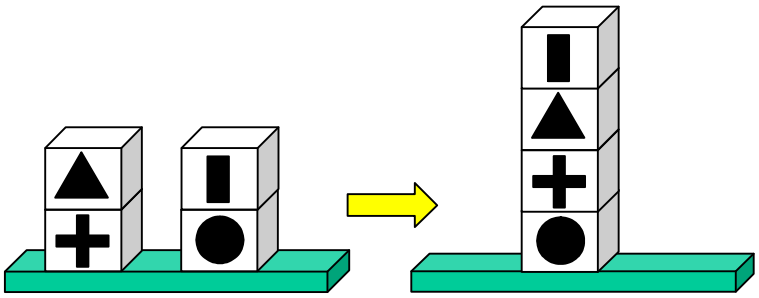
The planning server module is a wrapper component that implements the generic interface definition for planning tasks. The intelligent planning system in conjunction with other components is designed and integrated into a cooperative robotic system. This includes a vision server (a stereo camera system with pan-tilt unit), and a robot server. CORBA is used to integrate these different modules. These are located on different platforms, also are implemented into different operating systems. In order to integrate distributed systems, CORBA IDL (Interface Description Language) is defined as a planning system to connect and hide internal information to outside modules, i.e. wrap existing modules. In this thesis, an existing graphplan planning program is used with very few modifications. The graphplan module is wrapped by the IDL defined for the PLANNING server and seamlessly connected with other modules. The main task of the planner server is to provide a set of sequential actions that transform the initial state of a set of objects into a final or goal state.

D.1 Planning Systems

A Planning system is an intelligent system that generates a sequence of actions to achieve given goals [Russell and Norving, 2002]. There are many types of planning problems and approaches. Examples of planning problems are configuration planning solving for packing pallets into a truck, route planning in networks, path planning for robots.

To generate desirable actions to accomplish given goals, we need to describe a given problem. A *fact* describes a particular situation, i.e. which is the current state of a set of objects. An *action*, described by a *operator-object* duple, describes how this action changes the given facts after its execution. Before the action can be executed some *pre-conditions* must be true. A goal is a set of facts that should be true.

The *Block World* is a classic example of artificial intelligence research and we used it in this thesis. It consists of labelled blocks. In our experiments, a different shape is attached to each blocks, such as a circle and a triangle, instead of alphabetic characters. There is a manipulator which can perform the following actions: *pick up, put down, stack, and un-stack*. Unlike simulated experiments in artificial intelligence, in our experiment we actually use a robot manipulator that performs these actions with the help of visual feedback from a stereo camera. Figure D.1 shows an example of the block world with shape-labelled blocks. Figure D.2 shows an actual experimental setting, manipulator robot and block objects and stereo-camera unit.



(a) Initial State (b) Final Goal

Figure D.1: An example of the block world.

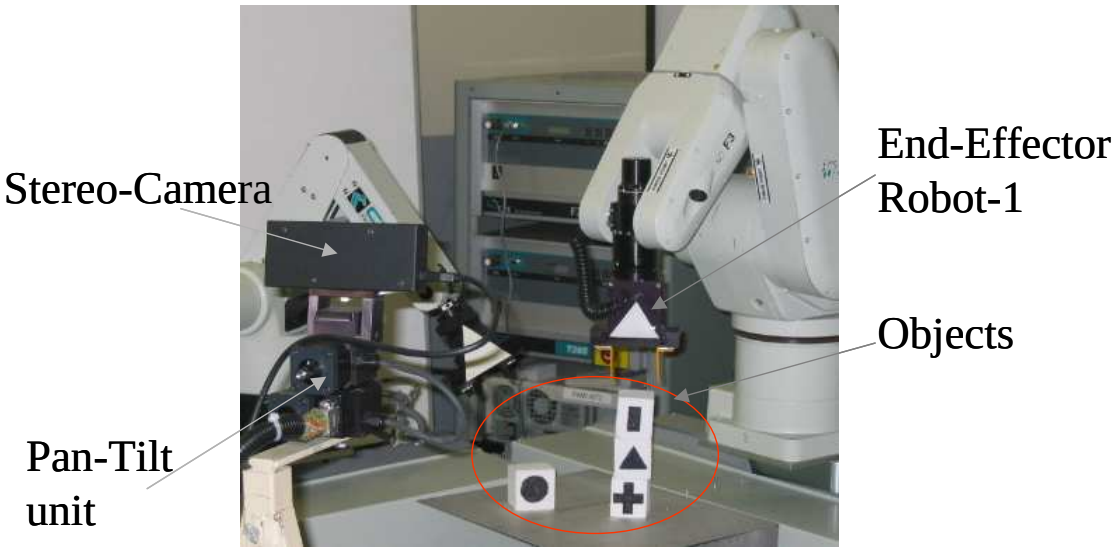


Figure D.2: Experimental set up.

One language used to represent a planning problem is STRIPS (Stanford Research Institute Problem Solver). STRIPS rules describe most of the common action of classical planning. They consist of *operators* (actions) with *pre-conditions* and *effects*. An *operator* can be executed if a set of *pre-conditions* (previous state) are true. After the

execution some *effects* change the current state. We use STRIPS type fact and operation descriptions. For example, *pick-up (x)* operator has the pre-condition of *on(x, Table)*, *clear(x)*, and *arm-empty*. The effect of *pick-up(x)* is *holding(x)*.

There are different types of planning systems. Among them we have Partial-Order-Planning (POP), Graphplan, and SATplan. POP uses least commitment search space. Graphplan exploits relaxed problem, then searches. SATplan translates to logic, and then uses satisfiability algorithms. In our research, we adapt Graphplan, because it always returns a shortest possible partial-order plan [Blum and Furst, 1997]. Some of the advantages of Graphplan are that it always finishes and is faster than POP. The main idea of Graphplan is to solve a relaxed problem. It converts to propositional representation. Then it constructs a graph with levels which has time steps. Graphplan, then, identifies simple inconsistencies between pairs of actions.

D.2 Fundamental state of Planning processing

In order to manipulate blocks, the following stages are performed: First, we need to identify each block's current position. This information is obtained from the vision server (See appendix C). Second, the actual physical position information is converted into a format that the graphplan planning module can understand. Physical location information is abstracted as to which block is on top of which block. Figure D.3 depicts these steps. Based on these facts, then the graphplan planning modules is asked to generate the sequence of actions so that the manipulator needs to follow in order to accomplish the required goal positions of the blocks.

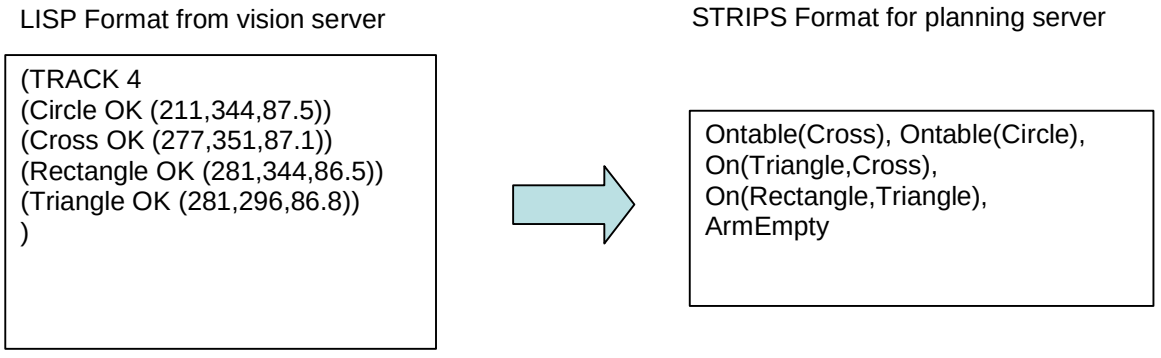


Figure D.3: Data transformation from vision server to planning server.

D.3 Main modules for Planning Server Wrapper component

Since our main research focus is to integrate multiple robots and build cooperative robots, we have decided to use an existing graphplan program. In order to use existing code and to reduce the complication of using the code, we designed a wrapper planner component. The wrapper component hides all the details of the planning system, and provides outside interface to other robots and modules.

As shown in figure D.4, to wrap the existing planning system program, we need to have two sub-components. The first sub-component is a data conversion component. It converts visual information and user commanded goal position information to the data that the planning system can understand. The second component is a data communication component. Then the outside module can ask for a planning operation and retrieve the actions that accomplish the planning goal.

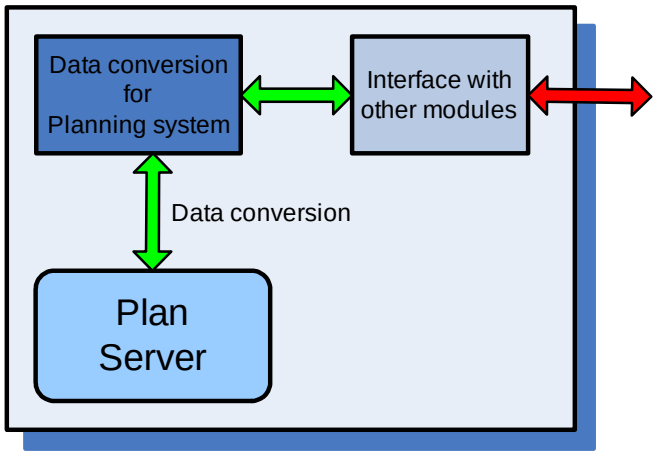


Figure D.4: Fundamental modules for planning server wrapper component.

D.4 Wrapper component CORBA IDL

We design CORBA IDL (interface design language) to implement a wrapper component for a planning system. Using this IDL definition, we have network stubs so that a server and a client can communicate through a network. CORBA IDL provides a basic network framework and we need to implement data conversion and task control module using IDL framework.

Next we design a "BLOCK" class, which parses the low level vision information and transforms it into a format that the planning system can understand. The next section explains how the information is composed into the IDL for interfacing to the planning system.

Planning system interface

Planning system is composed of two parts. One is the *server* part, which computes actual plan based on given "Operators" and "Facts". The other part is the *client*, which gets "Operator" and "Fact" inputs from files, from the user, or from other modules. First, Client reads the "Operator" and "Fact (goal)". It converts this information into Plan systems' own "GPLAN IDL sequence" (details will be found from the following section). Then it sends "GPLAN IDL sequence" to the Plan Server. The Plan Server reconverts GPLAN IDL sequence information into its own data list in order to compute a plan. Now, the Client can request a plan. The Planning Server computes a plan based on given Operator and Facts and returns the result. Finally, the Client can use the returned plan, and converts this high level plan into lower level, i.e. a more detailed plan in order that the robot can understand. In figure D.5 the summary of the planning system interface is described.

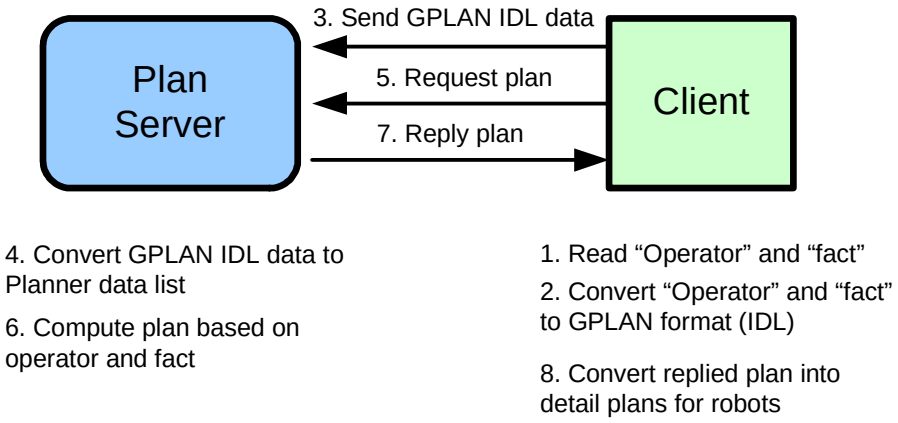


Figure D.5: Plan system interface and access mechanism.

GPLAN IDL interface

GPLAN IDL is made mainly for information passing between Planning Server and Client. In GPLAN IDL, there are three important things, one is opSeq sequence, which stores Operator data, another is Facts struct, which stores Fact data (goal), and the

other is planList sequence, which stores computed plan. When Client requests to compute a plan, it should send operator and facts using GPLAN IDL formats (opSeq sequence, Facts struct). Next the Planning Server returns a computed plan using the planList sequence. In Figure D.6 the complete IDL Planning interface is shown.

```
// Planner IDL

module GPLAN{
    struct Token    { string item;   };
    typedef sequence<Token> tokenList;
    struct Fact     {tokenList item; };
    typedef sequence<Fact> factList;
    typedef sequence<Fact> paramList;
    typedef sequence<Fact> precondList;
    typedef sequence<Fact> effectList;
    struct Operation
    {
            string name;
            factList param;
            factList precond;
            factList effect;
    };
    typedef sequence<Operation> opSeq;
    struct Facts
    {
            factList types;
            factList initials;
            factList goals;
    };
    struct Plans     {  string name;    };
    typedef sequence<Plans> planList;
    interface OpInterface {
            // read operation file and save into OP_LIST
            void getOperations (inout opSeq OP_LIST_clt );
            // read facts file and save into FACT_LIST
            void getFacts (inout Facts FACT_LIST_clt );
            // read facts file and save into PLAN_LIST
            void getPlan (inout planList PLAN_LIST_2clt);
            void startPlan ();
    };
};
```

Figure D.6: Planner IDL interface.

Bibliography

[Al-Mohamed *et al.*, 2003] Mayez Al-Mohamed, Onur Toker, and Asif Iqbal. Design of a multi-threaded distributed telerobotic framework. In *Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systesm, ICECS 2003*, volume 3, pages 1280–1283, December 14-17 2003.

[Arkin and Balch, 1997] R. C. Arkin and T. Balch. Aura: Principles and practices in review. *Journal of Experimental and Theoretical AI*, 9(2-3):175–189, 1997.

[Beni and Wang, 1991] G. Beni and J. Wang. Theoretical problems for the realization of distributed robotic systems. In *Proceedings of the International Conference on Robotics and Automation*, volume 3, pages 1914–1920, Sacramento, California, April 9-11 1991.

[Benmohamed *et al.*, 2004] Hcene Benmohamed, Arnaud Leleve, and Patrick Prevot. Remote laboratories: New technology and standard based architecture. In *International Conference on Information and Communication Technologies: From theory to applications (ICCTA 04)*, pages 101–102, Damas, Syria, April 2004.

[Bishay *et al.*, 1995] M. Bishay, M. E. Cambron, K. Negishi, R.A. Peters, and Kazuhiko Kawamura. Visual servoing in isac, a decentralized robot system for feeding the disabled. In *IEEE*, pages 335–340, 1995.

[Blum and Furst, 1997] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[Borstel and Gordillo, 2004] Fernando D. Von Borstel and José L. Gordillo. Modelo genrico y modular para desarrollar laboratorios virtuales en telerobótica. In *Proceedings of IEEE Computer Science Society, IX Ibero-American Workshops on Artificial Intelligence IBERAMIA 2004*, pages 399–408, Puebla México, November 22-23 2004.

[Bruyninckx, 2001] H. Bruyninckx. Open robot control software: the orocos project. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation*, volume 3, pages 2523–2528, Seoul, Korea, May 21-26 2001.

[Bruyninckx, 2002] H. Bruyninckx. Orocos: design and implementation of a robot control software framework. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, 2002.

[Casini *et al.*, 2003] Marco Casini, Domenico Prattichizzo, and Antonio Vicino. E-learning by remote laboratories: A new tool for control education. In *Proceedings of 6th IFAC Symposium on Advances in Control Education*, pages 95–100, Oulu, Finlandia, June 2003.

[Corporation, 1999] Microsoft Corporation. *Distributed Component Object Model (DCOM).* Microsoft Press, Redmont, Washington, 1999. http://www.microsoft.com/com/tech/dcom.asp.

[Deniz *et al.*, 2003] Dervis Z. Deniz, Atilla Bulancak, and Gokham Ozcan. A novel approach to remote laboratories. In *Conference of the ASEE/IEEE Frontiers in Education*, Boulder, CO, USA, November 5-8 2003.

[D'Souza and Wills, 1998] Desmond F. D'Souza and Allan Cameron Wills. *Object, components and frameworks with UML: a catalysis approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[El-Khalil *et al.*, 2004] Ibrahim El-Khalil, Insop Song, Federico Guedea, Fakhri Karray, and Yanquin Dai. Natural language interface for mobile robot navigation control. In *Proceedings of the 2004 IEEE Intl. Symposium on Intelligent Control*, pages 210–215, Taipei, Taiwan, September 2-4 2004.

[Esche *et al.*, 2003] Sven K. Esche, Constantin Chassapis, Jan W. Nazalewicz, and Dennis J. Hromin. An architecture for multi-user remote laboratories. In *World Transactions on Engineering and Technology Education*, volume 2, pages 7–11, 2003.

[García-Zubia *et al.*, 2005] Javier García-Zubia, Diego López de Ipiña, and Pablo Ordu na. Evolving towards better architectures for remote laboratories: a practical case. In *International Journal of Online Engineering*, volume 1, 2005.

[Giddings, 2006] Victor Giddings. Not your fathers corba - an architecture for embedded and real-time systems. *RTC Magazine*, October 2006.

[Guedea *et al.*, 2002] F. Guedea, I. Song, F. Karray, and O. Basir R. Soto. Multi-agent corba-based robotics vision architecture for cue integration. *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, October 6-9 2002.

[Guedea *et al.*, 2003a] F. Guedea, I. Song, F. Karray, and R. Soto. Enhancing distributed robotics systems using corba. *Proceedings of the first conference on Humanoid, Nanotechnology, Information and Control, Environment and Management, HNICEM-2003*, March 27-29 2003.

[Guedea *et al.*, 2003b] F. Guedea, I. Song, F. Karray, and R. Soto. Real-time feature extraction using interline method. *Proceedings of the International Symposium on Intelligent Control (ISIC' 2003)*, pages 626–629, October 5-8 2003.

[Guedea *et al.*, 2004] F. Guedea, I. Song, F. Karray, R. Soto, and R. Morales-Menéndez. Wrapper components for distributed robotic systems. *Proceedings of the Mexican International Conference on Artificial Intelligence (MICAI) 2004*, pages 882–891, April 20-26 2004.

[Guedea *et al.*, 2006a] F. Guedea, I. Song, F. Karray, and R. Soto. Building intelligent robotic systems with distributed components. *International Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII)*, 10(2):173–180, March 2006.

[Guedea *et al.*, 2006b] F. Guedea, I. Song, F. Karray, and R. Soto. Integration of distributed robotic systems. *International Journal of Advanced Computational Intelligence and Intelligent Informatics (JACIII)*, 8(1):7–13, January 2006.

[Henning and Vinoski, 1999] Michi Henning and Steve Vinoski. *Advanced CORBA programming with C++*. Addison-Wesley, Reading, 1999.

[Henning, 2004] Michi Henning. a new approach to object-oriented middleware. *IEEE Internet Computing*, 8:66–75, January-February 2004.

[Hexmoor *et al.*, 1993a] H. H. Hexmoor, J. M. Lammens, and S. C. Shapiro. An autonomous agent architecture for integrating "unconscious" and "conscious", reasoned behaviors. In *Proceedings of Computer Architectures for Machine Perception*, pages 328–336, Dec. 15-17 1993.

[Hexmoor *et al.*, 1993b] H. H. Hexmoor, J. M. Lammens, and S. C. Shapiro. Embodiment in glair: a grounded layered architecture with integrated reasoning for autonomous agents. In *Proceedings of The Sixth Florida AI Research Symposium (FLAIRS 93)*, pages 325–329, April 1993.

[Jacob, 2006] Joseph M. Jacob. Corba/e: Not your father's distributed architecture. *Electronic Design*, page 18, June 2006.

[Jia *et al.*, 2003] Songmin Jia, Yoshiro Hada, and Kunikatsu Takase. Development of a network distributed telecare robotic systems using corba. In *IEEE International Conference on Robotics, Intelligent Systems and Signal Processing*, pages 489–494, Changsha, China, October 2003.

[Li *et al.*, 2005] H. Li, F. Karray, O. Basir, and I. Song. An optimization algorithm for the coordinated hybrid agent framework. In *Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics*, volume 2, pages 1730–1735, Hawaii, USA, October 10-12 2005.

[Mecella and Pernici, 2001] M. Mecella and B. Pernici. Designing wrapper components for e-services in integrating heterogeneous systems. *The VLDB Journal*, 10:2–15, 2001.

[Musliner *et al.*, 1993] D.J. Musliner, E.H. Durfee, and K.G. Shin. Circa: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6), March 1993.

[OMG, 2000] OMG. Common object request broker architecture and specification (corba). Technical report, Object Management Group, Fall Church, USA, 2000.

[Pablos, 2004] Santiago E. Conant Pablos. *On the task-driven generation of preventive sensing plans for execution of robotic assemblies*. PhD thesis, Instituto Tecnológico y de Estudios Superiores de Monterrey, Campus Monterrey, Monterrey, N.L, December 2004.

[Pirjanian *et al.*, 2000] P. Pirjanian, T.L. Huntsberger, A. Trebi-Ollennu, H. Aghazarian, H. Das, S. Joshi, and P.S. Schenker. Campout: A control architecture for multi-robot planetary outposts. In *Proceedings of the SPIE Sensor Fusion and Decentralized Control in Robotic Systems III*, volume 4196, pages 221–230, Boston, MA, Nov. 2000.

[Pirjanian *et al.*, 2002] P. Pirjanian, C. Leger, E. Mumm, B. Kennedy, M. Garret, H. Aghazarian, S. Farritor, and P. Schenker. Distributed control for a modular, reconfigurable cliff robot. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, pages 4083–4088, Washington, DC, May 2002.

[Rumbaugh *et al.*, 2000] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Object Technologies Series, Reading, MA, USA, 2000.

[Russell and Norving, 2002] S. Russell and P. Norving. *Artificial Intelligence: A Modern Approach, Second Edition*. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[RWI, 1999] RWI. *Mobility 1.1, Robot Integration Software, User's Guide*. Real World Interface, Jaffrey, NH, 1999.

[Sanz *et al.*, 1999] R. Sanz, J.A. Clavijo, A. de Antonio, and M. Segarra. Ica: Middleware for intelligent control. In *Proceedings of the 1999 IEEE International Symposium*

on Intelligent Control/Intelligent Systems and Semiotics, pages 387–392, Cambridge, MA, Sept. 15-17 1999.

[Sanz *et al.*, 2001] R. Sanz, M. Alonso, I. Lopez, and C.A. Garcia. Enhancing control architectures using corba. In *Proceedings of the 2001 IEEE International Symposium on Intelligent Control*, pages 189–194, Mexico City, Mexico, 5-7 Sept 2001.

[Schlegel and Worz, 1999] C. Schlegel and R. Worz. The software framework smartsoft for implementing sensorimotor systems. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 1999*, volume 3, pages 1610–1616, Kyongju, Korea, October 1999.

[Schmidt *et al.*, 1997] D.C. Schmidt, A.S. Gokhlae, T.H. Harrison, and G. Parulkar. A high-performance end system architecture for real-time corba. *IEEE Communications magazine*, 35(2):72–77, February 1997.

[Simmons *et al.*, 1990] R. Simmons, L.-J. Lin, and C. Fedor. Autonomous task control for mobile robots. In *Proceedings of 5th IEEE International Symposium on Intelligent Control, 1990.*, volume 2, pages 663–668, Sept. 5-7 1990.

[Simmons, 2000] R. G. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, February 2000.

[Skubic *et al.*, 1995] M. Skubic, G. Kondraske, J. Wise, G. Khoury, R. Volz, and S. Askew. A telerobotics construction set with integrated performance analysis. In *Proceedings of the 1995 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, volume 3, pages 20–26, Pittsburgh, PA, August 1995.

[Song *et al.*, 2004] Insop Song, Federico Guedea, and Fakhri Karray. Distributed control framework design and implementation for multi-robotic systems: A case study on block manipulation. In *Proceedings of the 2004 IEEE Intl. Symposium on Intelligent Control*, pages 299–304, Taipei, Taiwan, September 2-4 2004.

[Trevelyan, 2004] James Trevelyan. Lessons learned from 10 years experience with remote laboratories. In *International Conference on Engineering Education and Research "Progress through Partnership"*, Ostrava, Australia, 2004.

[Utz *et al.*, 2002] Hans Utz, Stefan Enderle, Stefan Sablatng, and Gerhard Kraetzschmar. Miro - middleware for mobile robots applications. *IEEE Transactions on Robotics and Automation*, 18:493–497, August 2002.

[Wong *et al.*, 1999] Kwong K. Wong, Clive Ferguson, John Florance, Baliga Bantwal, and Trevor Jones. Flexible delivery of practical learning experience through the internet: The remotely operated cnc machine teaching project. 1999.

[ZeroC, 2002] ZeroC. internet communication engine (ice) home page. [online]. *Available: http://www.zeroC.com/ice.html*, 2002.