Arbitrated Robot Control on the Web, System Design and Implementation

by

Daniel Andrew Theobald

Submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements for the degrees of Bachelor of Science in Mechanical Engineering and Master of Science in Mechanical Engineering at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 1997

© Massachusetts Institute of Technology 1997. All rights reserved.

Author ..................................
Department of Mechanical Engineering
December 12, 1997

Certified by ....................................
J. Kenneth Salisbury, Jr.
Principal Research Scientist
Thesis Supervisor

Accepted by ....................................
Ain A. Sonin
Chairman, Department Committee on Graduate Students
Arbitrated Robot Control on the Web,
System Design and Implementation

by

Daniel Andrew Theobald

Submitted to the Department of Mechanical Engineering
on December 12, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Mechanical Engineering
and
Master of Science in Mechanical Engineering

Abstract

A new framework for controlling robots via the Internet has been designed and implemented in the Vision and Touch Guided Manipulation group at the MIT Artificial Intelligence Laboratory. This framework has been applied successfully to the problem of controlling the MIT Rover Test Bed- a small, tethered, wheel based mobile robot with two driven wheels in front, caster wheels in back, two passive cameras for stereo vision, and two arms. One arm is the first generation JPL Rocky7 rover arm on loan from JPL. The other arm is a four d.o.f. cable driven arm called the mini-WAM built by Thomas Massie as a small scale clone of the WAM robot built by Bill Townsend. The main features of the framework are it allows for great flexibility in the actual control strategy taken, and it allows transparent access to every level of the system through any computer connected to the Internet. Due to the fact that many "users" can access robot elements simultaneously, a priority based arbitration system is included in the framework to allow the system elements to filter conflicting inputs and operate in an orderly and effective manner. Potential ”users” range anywhere from a behavior process running on a computer in Japan, to a person using a GUI to access the robot from the 9th floor of the MIT AI Laboratory.

Thesis Supervisor: J. Kenneth Salisbury, Jr.
Title: Principal Research Scientist
Acknowledgments

At the end of the long road to thesis sits the tired student, grateful to the many individuals whom have contributed to this work, yet too weary to remember them all.

Thanks must go out to the many UROPs who have worked on the project, Sabina Ma and David Korka especially.

Thanks to Brian Anthony for all his help during my semester away and for making sure that I was not the last person in the office to finish my thesis. =)

Thanks to Ken Salisbury, my advisor for giving me the latitude to choose the direction of my research, and for providing the resources to carry it out.

Thanks to Eddy Baik for being a sincere person who cares about important things and constantly searches for the deeper meaning and purpose in himself, he is an inspiration.

A great big thanks must go to the most wonderful person in the world, Craig Lattimer. His friendship is priceless.

Thanks to my parents for raising me in a wonderful home and teaching me the timeless values that have helped me get this far, not merely in school but in life.

Most of all, I must thank my wife, for her unconditional love, support, and confidence, and the motivation to get this thing done.

Support for this work was provided by the National Science Foundation and the Jet Propulsion Laboratory.
Contents

1 Introduction .............................. 10
   1.1 Motivation .................................. 11
   1.2 Problem Statement .......................... 13
   1.3 Conventions used in this Thesis .......... 14
   1.4 Overview of the Thesis ................... 15

2 Overview of Previous Work ................. 16
   2.1 Web-Controlled Robotics .................. 16
      2.1.1 The Mercury Project and the Telegarden. .. 18
      2.1.2 Perth .................................. 19
      2.1.3 Berkley’s Mechanical Gaze ............... 21
      2.1.4 Robotic Telescopes ...................... 23
      2.1.5 JPL’s Rocky7 ........................... 24

3 Design of the Arbitrated System Control Environment 26
   3.1 Overview .................................... 26
      3.1.1 Design Space and Design Goal .......... 27
      3.1.2 List of Design Issues ................... 28
   3.2 Processing ................................... 29
      3.2.1 Possible Solutions ........................ 29
      3.2.2 Design Requirements .................... 31
   3.3 Computing paradigm ........................ 31
      3.3.1 Possible Solutions ........................ 31

4
3.3.2 Design Requirements ........................................ 33
3.4 Documentation .................................................. 33
3.5 Network accessibility ............................................. 34
  3.5.1 Should the system be web accessible? ....................... 34
  3.5.2 Possible implementations .................................. 36
  3.5.3 Design Requirement ........................................ 41
3.6 Robust error handling and Debuggability .......................... 41
  3.6.1 Possible Solutions ......................................... 42
  3.6.2 Design Requirements ....................................... 44
3.7 Summary of Design Requirements ................................ 44

4 ASCE Implementation ................................................. 46
  4.1 Overview ....................................................... 46
    4.1.1 Why Java? ................................................. 46
  4.2 The Mirror Subsystem .......................................... 48
    4.2.1 The mirror model ....................................... 48
    4.2.2 How mirrors work ....................................... 50
  4.3 The Arbitration Subsystem .................................... 54
    4.3.1 The Arbitration Model .................................. 54
    4.3.2 How Arbitration Works .................................. 55
    4.3.3 Shortcomings of this implementation .................... 56
  4.4 The Port Subsystem ............................................. 59
    4.4.1 How Ports Work ........................................ 60
    4.4.2 Issues in Process Flow .................................. 62
  4.5 The ServoLoop Subsystem ...................................... 63
    4.5.1 Fault Tolerance ......................................... 64
    4.5.2 Real-Time Support ....................................... 64
  4.6 The Behavior Subsystem ........................................ 65
    4.6.1 How Behaviors Work .................................... 65
    4.6.2 The Handling of Mirrors ................................ 66
C.4 asce.arbitration.Arbitrator .......................... 112
C.5 asce.port.InputPort ..................................... 113
C.6 asce.port.OutputPort ..................................... 116
C.7 asce.port.PortManager .................................... 116
C.8 asce.port.PortOwner ...................................... 117
C.9 asce.behavior.Behavior .................................. 117

D Where to find more information. .......................... 119
D.1 Online docs for the RTB system .......................... 119
D.2 Online docs for the Java language ....................... 119
List of Figures

2-1 Shown here is the famous Internode Systems SNMP Controlled toaster. 17
2-2 The Mercury Project at USC let web users move the robot and blow
air to uncover clues in a sandbox. .................................. 19
2-3 The Telegarden allows web users to plant seeds and water them and is
currently on display in Ars Electronica Center. .................... 20
2-4 The Perth Robot at the University of Western Australia. .......... 20
2-5 Shown here is a screen shot of Perth’s Web User Interface. ....... 22
2-6 Here is Perth placing a block under command of the author. ....... 23

3-1 The merging of top-down and bottom-up design modes. .......... 27
3-2 An illustration of the Mirror Paradigm. ................................ 39

4-1 Shown here is a possible mirror tree. Notice that there is no conflict
with having two mirrors for the same root running on the same com-
puter as long as they have unique names. In general this is not a very
efficient way to do things, however. ............................... 49
4-2 A more efficient layout for the mirrors. ............................. 50
4-3 An arbitration example. ............................................. 57
4-4 An example of external or public ports vs. internal or private ports.
The BaseEncoders and BaseMotors objects are shown inside the Sim-
pleBaseController because it abstracts them. ..................... 61
5-1 The rover before the miniwam was mounted. The two sphere were used in a behavior that would visually servo the rover to a lander mockup using the overhead camera.

5-2 A close up of the drive system.

5-3 The JPL arm.

5-4 The mini-WAM before it was mounted on the rover.

5-5 The rovers cameras.

5-6 Some of our Computers.

5-7 A layout of the RTB System. Note that only a few components are shown, dashed round rects represent object mirrors.

6-1 One run of the Behavior Arbitration and Port usage experiment.

6-2 There was obvious human involvement here. One incarnation of the SimpleBaseControllerGUI is shown.

6-3 The path of the simulated rover in a walled-in test. This shows several simple elements working together to create complex behavior.

6-4 A Walled-in test with the RandomTurnBehavior disabled.

6-5 Shown here is the view from the left eye as the rover visually servos on the green ball. The buttons at the bottom are for selecting the channel and choosing the color to look for.

6-6 Shown here is the rover as it visually servos on the green ball.

6-7 Shown here is the state of the BaseMotors object just before the AvoidObstacleBehavior takes over.
Chapter 1

Introduction

This thesis presents the design and implementation of a new framework for controlling robots via the Internet. This framework has been applied to the problem of controlling the MIT Rover Test Bed- a small wheel based mobile robot with a simple vision system and two arms. The main features of the system are that it allows for great flexibility in the actual control strategy taken,(eg. centralized, distributed, behavior-based, etc.), and that it allows transparent access to every level of the system through any computer connected to the Internet. Due to the fact that many computers can access robot elements simultaneously, a priority based arbitration system is included in the framework to allow the system elements to filter conflicting inputs and operate in an orderly and effective manner.

Most current web-based robotic systems have a very clear line where the robot system ends and where the user-interfaces begin. This line is the web server that interprets commands received from remote users on the internet and issues commands to the actual robot. An analogy of this type of system is a radio station playing requests that listeners call in. The framework developed in this thesis blurs and maybe even eliminates that line by allowing remote “users” to interact with every level of the system, allowing them to actually become part of the system, rather than only allowing access to a graphical user interface (GUI) which has limited interaction with the rest of the system. The intent of this thesis is to contribute this new approach to the body of knowledge dealing with web-based robotics.
This chapter contains a section on the motivation for this research presented anecdotally in the first person, a section with the problem statement, and concludes with an overview of the remainder of this thesis.

1.1 Motivation

This thesis comes as the result of my desire to create a robot programming environment with more flexibility, ease of use, maintainability, accessibility, and stability than what I had been using in the past.

When I started working in the MIT Artificial Intelligence Laboratory, I began working on the Whole Arm Manipulator (WAM) used in conjunction with the Fast Eyes Gimbals (FEGs) to grasp cylindrical objects lying on the floor. The arm and eyes were controlled by a VME-bus based system running a one of a kind operating system developed by Gunter Niemeyer called Hummingbird. This system has been able to do some very impressive things such as catch a free-flying tossed ball, and more recently, catch paper airplanes.

While this system was very fast and could perform these impressive feats, one had to be a skilled computer hacker to program the system to perform new tasks. This was in part because the system had evolved over several years under the guidance of different programmers. It was fairly complicated, involved several different types of hardware, and lacked in-depth documentation.

I do not mean to imply that things had not been done correctly on that system, for this would be very far from the truth. Two of the main programmers who have worked on the system over the years Gunter Niemeyer, and “Jesse” Won Hong, are easily among the most talented engineers I know. They knew the system inside and out, and in an evening of hacking could do marvelous things. The problems arise when a neophyte to the system comes along and tries to start programming it.

A critical point to note about the WAM system, is that in fact it is a research project, not an industrial project. The latter is a very different beast, often with huge staffs to design and implement standards, review code implementations, maintain past
code and produce future releases. University research, on the other hand, is made up of researchers often with a handful of over-worked grad students trying to push the limits of technology in very specific areas. Trying to get a demo working to show some new capability or a proof of concept to funding sources is the par. At that point no one really cares about the maintainability and extensibility of the system, they only care about whether it works or not.

When I was given the opportunity to put together a robotic system from scratch, I took it as an opportunity to try to design and implement a system with some of the features that I had longed for during my previous work. I wanted to create a system that was easily programmable, highly documented, modular, and easily extensible. In the summer of 1996 I co-authored a paper called “Autonomous Rock Acquisition” for an AIAA conference in August 1996. In this paper I presented some of my ideas for the design of a hybrid system combining benefits of both distributed control and centralized control which I called “Augmented Distributed Control.” I presented a list of desirable qualities for that system. That list follows:

1. Complete autonomy in the execution of very high level commands.

2. Robust error handling and recovery built in.

3. Easy to program, natural task representation.

4. Abstraction of detail, hardware independent.

5. Able to carry out useful work even in the absence of high level commands, eg. default behaviors.

6. Able to carry out tasks in parallel.

7. Able to handle redundant sensors naturally.

8. Able to handle redundant processors naturally.

9. All elements allowed access to a centralized world model.

10. All elements have a standard interface to the rest of the system.
11. Elements can come on-line or go off-line in real time.

12. Allows for centralized control when desired.

Now, nearing the end of (the first phase of) this project, I am pleased to realize that the system I have designed and implemented addresses every one of those desirable qualities with only slight modification. (Specifically, the centralized world model became an element which could exist as part of the system, but is not required.) I will discuss this in conjunction with other approaches taken in robot control in the next chapter. As the project progressed, I began to see the potential of using the web to help accomplish some of these original goals and steered the project in that direction.

1.2 Problem Statement

This thesis most neatly falls into the category of a system design project. The intent was to develop a framework, mostly in the form of interfaces and helper classes, that will, in the future, be of use to people working on the MIT Rover Testbed (RTB) or other robotic projects.

The first step taken was to come up with a specific list of desirable attributes, or more formally, design requirements. This step included a large amount of design exploration as well as significant trial and error in test implementations using different programming languages, hardware, etc. The next step was to come up with a possible design solution that met the established design requirements. This, as every designer knows, is an exercise in trade-offs and compromise. Following this decision making process a working implementation of the framework system was created. The framework was then tested by using it on a real robotic system, the MIT Rover Testbed, where sample controllers and behaviors where implemented using the framework.
1.3 Conventions used in this Thesis

Due to the large number of acronyms and technical words used in this thesis, a glossary has been included in Appendix A, and an effort has been made to include all words that might be found troublesome to the reader without background in robotics and Java programming.

The code for this thesis was written primarily in Java, and hence some explanation of Java conventions would be useful. The Java language, developed by Sun Microsystems, is an object oriented system similar in many ways to C++, but also very different. Java is both a compiled and an interpreted language. Java code is first compiled into bytecode, a machine independent code, and then the bytecode can be run by an interpreter on many different operating systems. Java has a strict convention on the naming of classes and interfaces, which should be included in packages. A package is a grouping of classes and interfaces with a common purpose. Each class or interface must be defined in a separate file carrying the name of the class it defining, and that file must be stored in the correct directory for its package. For instance, “asce.mirror.Mirrored” is the full package name for the Mirrored interface which is defined in a file called Mirrored.java which is saved in the “/asce/mirror” directory. If this seems unclear, there are several books on Java which explain it in detail.

Java classes will generally be referred to by their full package name to avoid any possibility of confusion. Methods (functions within a class or interface) will be referred to by the class’s full package name followed by a colon with the method name unless the class is clear from context, (i.e. asce.mirror.Mirrored:stateChange). Most of the references to classes and methods will point the reader to the corresponding code in the Appendix C.

The word “user” will refer to both humans and other objects or processes which might make use of a particular system element. For example it is possible for either a behavior running on a computer or a human operator to take control of part of the robot.
1.4 Overview of the Thesis

Chapter 2 explores some of the projects that have been done in the area of web controlled robotics. It discusses the general strategies taken, and how the work in this thesis differs.

Chapter 3 presents the major issues that were considered in the design of the Arbitrated System Control Environment (ASCE). Its sections explore each issue in detail, offering several possible solutions and choosing one for implementation. The last section in chapter 3 summarizes this information with a concise list of design requirements and corresponding design specifications.

Chapter 4 presents the interfaces and subsystems implemented to meet the design requirements presented in chapter 3. Each subsystem is presented in turn, followed by an integration section presenting an example of how they all work together as a complete framework for web-based robotics. In addition, chapter 4 has a brief discussion of why Java was chosen as the primary language in which to implement the framework.

Chapter 5 presents the MIT Rover Test Bed (RTB) in detail and discusses the application of the ASCE framework to it. The first part of the chapter gives an overview of the hardware, and the remainder is devoted to explaining the java package written to run the robot which is named rtb (lower-case package names are another Java convention).

Chapter 6 presents some sample behaviors that have been implemented using the system, mentions some issues that haven’t been dealt with directly, suggesting possible solutions, and concludes with some recommendations of behaviors that would be useful or interesting for future research.

Chapter 7 presents the conclusions and some suggestions for future work.

The appendices contain a glossary of terms and acronyms used in this thesis, a mirror programmers guide for additions to and maintenance of the system, various code listings referred to in the text, and a list of sources for more information on Java and the ASCE framework.
Chapter 2

Overview of Previous Work

The first part of this chapter explores some of the projects that have been done in the area of web controlled robotics, the remainder of the chapter is devoted to a discussion of previous work in the area of robot control strategies.

2.1 Web-Controlled Robotics

Probably due to the great success and popularity of the internet over the past few years, we have seen the birth of a new type of mechanical device. These new devices might be best classified in the broad category of web-controlled robotics. The first such devices were pure novelty, however, this class of device seems to have left the embryonic stage and is now in very early infancy as demonstrated by some of the newer telescope and rover projects. These newer projects take the novelty of web-controlled robotics and turn it into a tool for sharing scarce scientific resources with a vast global audience.

Some of the early and more well-known web-controlled robotic devices are listed:

1. The Mercury Project and the Telegarden at the University of Southern California.

2. The Perth Telerobot in Australia.

3. The Mechanical Gaze at Berkley.
Figure 2-1: Shown here is the famous Internode Systems SNMP Controlled toaster.


5. The Rocky7 rover at the Jet Propulsion Laboratory.

While there are probably many other systems out there, both on-line and under development, the intent here is to show some examples of related work and a broad view of the current state of web-based robotics. For interests sake, other than the systems listed above, a web-search turned up “The Internet Toaster”, various remote cameras, and a web controlled model railroad in Germany. The Internet Toaster (Figure 2-1) is claimed to be the first web-controlled mechanical device. While making web-toast may not seem like a particularly useful thing to do, it was demonstrative of some of the possibilities of the future internet.

All the systems found to-date are very similar in that they have some way to command the robot’s new position, and provide still-frame visual feedback once the move has been made. All systems present the user with some sort of GUI to control the robot, but there is no opportunity to retrieve lower-level data from the robot, program the robot, or extend the system. This is the area to be addressed by this thesis. The difference is mainly that these systems are set up such that there is a robot with a control system, and then a subsystem is added to allow high-level control via the web. On the other hand, the system developed in this thesis is completely integrated with the web, and the web is essentially part of the robot.
2.1.1 The Mercury Project and the Telegarden.

The Telegarden is the direct offspring of the Mercury Project, which was one of the earliest truly interactive web-robots. It was conceived and developed at USC, both as a novelty, and a way to study how the internet community interacts with each other. The robot used was an old industrial robot that they had in the lab. The goal was to create a system which the internet community could actually use to modify the environment in some way. However they found it difficult to conceive a system that would allow for only non-destructive interaction with the environment.

Eventually they came up with the idea to remove the robot’s end-effector and, replace it with a pneumatic supply hose which could give a burst of air when desired. The workspace was filled with sand, and clues to a story where buried in the sand. A picture of the robot appears in Figure 2-2. Users were then able to take turns moving the robot, blasting some air, and seeing what, if anything, was buried in the spot they had chosen. There was a newsgroup provided where people could post what they had found, and share their theories about what the clues mean. The site was very popular making it difficult at times for users to get on the system.

The success of the Mercury Project inspired the designers to create a new web controlled robot which would offer more opportunity to study how the world-wide internet community interacted. In the Telegarden (http://www.usc.edu/dept/garden/), the robot’s workspace was filled with dirt. The robot was able to plant seeds by blowing a hole in the dirt, then dropping the seeds in, as well as water the seeds and plants in the workspace. See Figure 2-3. They have had to wipe out the garden and start from scratch on several occasions, due to the fact that everyone wants to do something, and plants only need so much water. They have addressed that problem in part by limiting the amount of water used.

The questions raised here are, (1) how do you know that Joe User’s intentions are constructive rather than destructive, (2) how to limit the amount of damage possible if they are destructive, and (3) how to provide enough information about the current state of the garden so that he can make good decisions if his intentions are good. They decided from the outset of the project that if anything were to be learned from
the project that there would have to be the possibility of failure. This is because if success is guaranteed, e.g. if the system kept track of how much water each area needed, and only allowed that much, as well as watered those areas itself that where neglected, then the community becomes entirely unnecessary. Obviously eliminating system dependence on the variable of interest (the community) does not facilitate study of that variable.

2.1.2 Perth

The Perth telerobot is located at the University of Western Australia, and can be found on the web at:

http://telerobot.mech.uwa.edu.au/.

It consists of an industrial robot positioned over a table with wooden blocks. The author was able to log on to the Perth system and control the robot for about a half an hour. A color image of the robot is presented in Figure 2-4. There were
Figure 2-3: The Telegarden allows web users to plant seeds and water them and is currently on display in Ars Electronica Center.

Figure 2-4: The Perth Robot at the University of Western Australia.
approximately 600 sessions per week in 1996. The robot allowed a single user to take control at any one time on (what seemed to be) a first come first serve basis, although preference was given to people who filled out their online survey.

Once a user had control of the robot, s/he was presented with the user interface shown in Figure 2-5. It allowed for simple control of the robot by specifying the new x,y,z,tilt and spin coordinates of the end effector, and whether the gripper was opened or closed. The other option was to specify sequential moves, e.g. "open z10 close z100 y70 open." It was a rather unnatural and slow way to carry out a task, but with a little effort the author was able to successfully stack three blocks on end as shown in figure 2-6. Later the same day, the robot was found in a state where joint 6 was at its limit. After quite a while of trying to back away from the limit, the author conceded defeat. It seems that the robot is programmed with restricted movement in cartesian space, as it will not accept a command to move the end effector out of a 500X500X450 unit cube. However, there is no guard against moving to joint limits, and hence the robot can get stuck in a position which is difficult to back out of.

Most likely due to the difficulties of controlling the robot by typing in numbers, Perths caretakers are developing a Java applet which shows graphically where a given command will move the robot. It is assumed that eventually they will be able to drag the graphical end effector to the desired position which then commands the actual robot to the same position. This would be an example of a predictive display. Note that this applet is being added to the user interface as a visualization tool, and is not a direct part of the robotic system. In the system developed in this thesis, the goal is to provide for simple integration of such components directly into the system.

### 2.1.3 Berkley's Mechanical Gaze

Berkley's Mechanical Gaze can be found at http://vive.cs.berkeley.edu/capek/. Even though the web page claims continuous operation since March 1995, they have been moving the robot to a new location, and hence it has been unavailable for exploration. The idea of the robot is to provide a camera mounted on the end effector that can be used over the web for exploring a remote space. The project engineers set up a
Figure 2.5: Shown here is a screen shot of Perth’s Web User Interface.

Control robot

Enter a set of moves (example "open z10 close z100 y75 open"): [Text Input Field]

or specify only the next position and orientation of the gripper


Relinquish control? [□ Confirm]
small museum display with interesting objects to look at. In scanning through the comments there seemed to be many people who had trouble controlling the robot, or getting a picture, but others found the robot to be very usable and interesting. Email had been sent to the engineers asking for information on challenges they have had and things they have learned, but they had not responded at the time of this printing.

2.1.4 Robotic Telescopes

There are currently a few different robotic telescope projects out there either in operation, or under development. These are in the most basic case servo-controlled two degree-of-freedom telescope pointers coupled with a camera. The idea here is that not everyone can afford a full scale telescope of their own, but that there are many telescopes out there that are only used a small fraction of the night hours in which they are useful. One way to make better use of the resource is to allow people
to go to the site and use the telescope in what would otherwise be downtime. This requires that people travel to the site which is expensive and time consuming, and also requires people to conform their sleep schedules to the available observatory time. The better option (now that it is possible) is to create a web-based interface which allows remote users to request and schedule use of a telescope, and retrieve the images via the web.

One such system is the Bradford Robotic Telescope Observatory, which can be found at http://www.telescope.org/rti/use.html. It claims to be “The first of its kind in the world.” Information presented on the web site contains the following explanation, “The Bradford Robotic Telescope is 46cm and totally autonomous. Located high on the moors in West Yorkshire, England. The telescope decides when the conditions are good enough to make observations of the sky by itself (an astronomer does not need to be present). Anyone on the Internet can register and ask the telescope to look at anything in the northern night sky. Observations are automatically prioritised and scheduled and completed by the telescope as time allows. Other data (weather information and reports) are obtained and updated on this site automatically every day. The telescope and this system are prototypes. We are currently working with a number of schools and institutions to provide telescopes similar to this around the world.” Currently the Bradford Robotic Telescope allows anyone on the internet to make use the telescope on a first come first serve basis and the system can handle a maximum of 160 simultaneous web requests.

One of the predominant research issues involved in the web-based-telescope concepts deal with optimum scheduling of resources. For example, if there are many requests for a single night with various targets, what trajectory should the telescope follow through the night to satisfy as many of the requests as possible.

2.1.5 JPL’s Rocky7

Interestingly the Rocky7 project at JPL turned to the web concurrently and independently from the project at MIT. The approach taken, however, was necessarily different. The web interface is a collaborative development between Jet Propulsion
Laboratory (point of contact: Paul Backes) and IA Tech, Inc. (point of contact: Kam Tso). The Web Interface for Telescience (WITS) is a subsystem that was added to the original system to allow users of the web to use the robot from their home institutions rather than having to travel to JPL. It is not directly “connected” to the rover as it requires that the engineering team uplink the final plan to the rover. In contrast, the approach taken in the MIT project was to create an experimental system completely connected to and integrated with the web, allowing for the easy creation of user interfaces and hardware sharing as part of the basic system rather than adding these amenities as subsystems.

The WITS user interface is available on the web at:

(http://mars.graham.com/rocky7wits/index.html).

It runs a simulation of the rover as it is mainly for public outreach, and therefore does not have to deal with issues of unauthorized use and prioritized distributed control. In the mission version which will actually be used to command the robot during tests or on Mars, user access is controlled by IP address and password protected user login. All users who log into WITS have a user type: Scientist, Navigator, Viewer, or Mission Planner. Scientist, Navigator and Mission Planner users can modify the common database at JPL which contains plans for future rover tasks, and all users can see what other users submit to the plan. Currently it is up to the principal investigator to decide which requests are put in the plan and provide the final plan to the engineering team for uplink to the rover.
Chapter 3

Design of the Arbitrated System Control Environment

3.1 Overview

This chapter presents the major issues that were considered in the design of the Arbitrated System Control Environment. The balance of this section contains some comments on the design process in general, a definition of the design space, and a list of the issues considered. The following sections explore each issue in detail, offer several possible solutions and choose one for implementation. The last section summarizes this information with a concise list of design requirements and corresponding design specifications.

The growth and development of this system specification or design can best be modeled as an evolutionary process rather than a strict design methodology. It cannot be classified as purely top down, or bottom up. It can most accurately be described as a bottom-up idea - implementation - revision cycle in parallel with a top down idea - implementation - revision cycle, the two generally meeting somewhere in the middle. Some of the top down elements end up filtering all the way through the system to the bottom, and some from the bottom to the top. A graphical representation of this process is shown in Figure 3-1.

This iterative design method is by no means the most efficient way to “engineer”
the resulting system, however, this is a research project and hence the exploration of new technologies and new ways of doing things has taken precedence over finding the most direct and cost effective way to get from point A to point B. For instance, the use of Java as the main programming language of this system, even for the low-level controllers, might not make sense from the standpoint of creating the best control system. On the other hand, it provides a convenient way to create a homogenous system which facilitates the meeting of other design requirements such as providing runtime access to every level of the system. More on the justification for using Java will be presented at the beginning of the next chapter. The point to be made here is that the design tradeoffs made in this project are most likely very different from those that would be made for the same project in an industrial setting. Simply put, the focus here is exploration and not necessarily a finished product.

3.1.1 Design Space and Design Goal

Determining the design requirements can be vague and confusing without a well defined design space. The design space for this project consists of one to several computers working on a high-level common goal (or set of goals) possibly with specific
hardware connected to individual computers. The design space is further divided temporally in two subspaces: programming-time, and run-time. Programming-time includes time spent on maintenance, debugging, and extending the system. Run-time is what the system should be capable of doing. The design goal for this project is an error-tolerant system that is easily accessible, extensible, and maintainable, and capable of carrying out complex high level goals. A test platform for the design implementation is the MIT Rover Test Bed.

3.1.2 List of Design Issues

The following is a list of various issues, options, and problems that have become apparent in reaching for the design goal in the design space. Each will be discussed in the following sections. These issues tend to be very inter-related, and as always, this forces design tradeoffs. The tradeoff debate could resemble the following dialog. “The system should run fast, the system should be object oriented, and highly general. However, generality often precludes optimum efficiency and therefore it must be asked how fast is fast enough and how general is general enough?” Many of these questions can only be answered in the qualitative realm of opinions based on personal experience where a typical decision will be to sacrifice a certain amount of speed in favor of generality and visa-versa. The issues specific to this project follow. They will be discussed in reference to the general design goal, and occasionally comments will be added relating to the specific system requirements of the MIT Rover Test Bed.

1. Processing: How to meet the processing requirements.

2. Computing paradigm: What general approach should be taken? Linear programming, object oriented, text-only, or graphical?

3. Documentation: How to make sure it is well documented for future use and modification.

4. Network Accessibility: What level of network access should be allowed and how to achieve it.
5. Robust Error Handling and Debuggability: How to avoid having one error bring the system down, and to make it easy to track down errors and fix bugs.

The remainder of this chapter will explore several key design issues, and present some main points of the evolution of each design requirement.

3.2 Processing

The first question to be asked when doing a computer-related project of this type is “what type of computing system will be used to do the processing?” The answer to this question unfortunately tends to determine many of the future design decisions because of the capabilities and limitations of the platform or system chosen. Another reality of choosing processing options is that while computers are very cheap compared to what they have been in the past, they are still not “a dime a dozen” and hence new projects must often make use of whatever computers happen to be available.

Another issue is how to add more hardware to the system? Suppose that a robotic system is setup on a VME bus, and the project needs another frame grabber, or controller, but all the slots on the VME bus are filled. It would be ideal to be able to add hardware as needed, allowing the system to grow and scale naturally without having to recreate the entire system every time a limit of the current setup is encountered.

3.2.1 Possible Solutions

With these issues in mind, the ideal solution seems to be to design a system that can make use of whatever processing is available. In other words, a platform independent system. Of course, complete platform independence is not currently possible whenever a computer is used to control specific hardware, as device drivers specific to that computer are needed. However it is possible to create a level of platform independence by creating device drivers that have a standard interface to the higher levels in the system. Using this approach, the majority of the code that is written can be platform independent using any standard language that can be compiled on most computers.
Of course, platform independence is taken to an even higher level with an interpreted language that does not need to be recompiled for each computer, languages such as Lisp, or Java.

The issue of adding new hardware to the system brings up the question of how to communicate between the different parts of the system. The only way to provide for total flexibility, as well as hardware independence, is to connect the processors through a standard that any type of computer can connect to. Such a standard is Ethernet using TCP/IP. Using a networking solution such as this has several tradeoffs. Currently, it limits the ability of the system to directly share memory or to have high-bandwidth communication. However this is not a real problem for a system that operates more in the range of distributed control rather than centralized control. Also, because the specific system we are designing for here, the MIT Rover Test Bed, is a system that will interact with an essentially static environment, there is no compelling reason to sacrifice system generality for speed of inter-node communications. In addition, specifying that the system nodes communicate via Ethernet by no means precludes the possibility of adding subsystems such as a VME bus, and in the most general case allows for the inclusion of every computer on the internet in the system.

The discussion of the communication protocol could go much deeper here as there are different standards for sockets that handle lost packets and error checking differently. When dealing with a tele-operated robot, in general it is not advantageous for the robot or controller to request that a lost packet be resent because by that time the information will be old and not nearly as useful as if the latest information were transmitted instead. However in a general system such as is being designed here, there are many cases when a single message could be very important and therefore should not be dropped. Probably the best system would allow for both types of connections to be made. However, for system simplicity, it is easier to simply use a standard that guarantees the (eventual) arrival of every packet sent, or generates an error if the connection is lost.
3.2.2 Design Requirements

The design requirements arrived at in this section are that the system be programmed primarily with platform independent code and that processors or computers are connected using standard ethernet using some standard socket protocol.

3.3 Computing paradigm

Should the system be object oriented? This is a question with clear tradeoffs. On the one hand object oriented languages are typically slower than non-object oriented languages. On the other hand, object orientation brings with it many of the other design requirements discussed in this section. For example, object oriented languages in general make it easier to write code which is reusable, maintainable, extensible, and debuggable. In addition some offer convenient documentation aids which greatly simplify the documentation process.

3.3.1 Possible Solutions

It seems clear that object oriented is the way to go. In places where speed becomes critical, there is usually the possibility of writing that bit of code in C or Assembly, and then linking it into the rest of the code. Once the decision to use an object oriented paradigm is made, the next question is how to represent the system symbolically, and to what level of detail.

One option is to write the code with an object to represent each physical element of the system. For example, an object to represent the robot’s arm, one for the base, one for the cameras. One benefit of this approach is that it allows the possibility of creating high level commands which abstract away all the detail of implementation. For instance, I could ask the cameras object if there is a ball in the workspace by calling a method like ball = findBall(). If findBall returns with a ball I could then call a gotoBall(ball) on the base object followed by a grabBall(ball) method on the arm object. This provides an easy way to implement high level behaviors, and lends
itself towards the implementation of a scripting language approach. The scripting language idea is nice because it provides a way to script out new sequences of actions without having to write and compile any new code.

One drawback to the method just explained is that implementing the individual objects becomes a difficult task because of their size and complexity. For example, the base object needs to have methods to access the encoders, and to output currents to the motors. It needs to implement a controller, a trajectory generator, a trajectory spooler, and probably some kind of navigation. Another complication is that at different times, it might be desirable to run the base in different modalities. For instance one might need to be able to navigate both in absolute and relative frames. For example:

1. goto(x,y,theta) meaning move the base to absolute position (x,y) pointing in direction theta.

2. moveForward(x) meaning move forward relative distance x.

3. moveWheels(leftWheel, rightWheel) meaning move the wheels to the absolute angles specified.

These are just a few of the commands which might come in handy for navigation purposes, and they are very different in their implementations. The first requires inverse kinematics computed for the base to figure out how to get to the point specified by driving the wheels. This non-holonomic problem can be quite complex and involved especially when things are added such as obstacles.

Another drawback becomes apparent when thinking about modifying or extending this object. First of all, this would be impossible in real time while the system is running. It would require bringing down the system, writing all the new code, and then starting it back up. In addition, adding new methods or functionality to an object of this size becomes a problem because it is not only possible but likely that the currently functioning object will in some way be broken. Changing some aspect of the controller code may very easily break code that was tuned to how the controller used to respond. Maybe it would be possible to write new controller code while
leaving the old intact, and providing some way to switch between the two. While this is a possible solution, it adds additional unnecessary complication. However, it does hint at another possible solution to the problem: decreasing the grain size of our objects.

Another design option would be to have computing objects representing a finer grain size of the physical system. For instance, instead of having one large object representing the base, there could be an object representing the base motors, one for the base encoders, one for the controller, etc. One could then create a base object which provided the type of interface explained above by incorporating these sub-elements into it, essentially wiring them together for a particular function. However, it would not preclude the programmer from making use of the lower level elements directly when desired. This creates a multilevel scaleable environment where the programmer or user has access to elements on whatever level is necessary, while unnecessary detail is abstracted away by the higher level groupings of objects.

3.3.2 Design Requirements

The design requirement is a scaleable component based system. For example, detail should be hidden when desired by working with the Arm component as a complete entity. When it becomes necessary to delve into more detail, you can get a list of components contained in Arm, and deal with them directly. This list might look something like this: ArmEncoders, ArmController, ArmInvKinematics, ArmMotors, ArmTrajPlanner, etc. Each of those in turn could contain sub components or be a base component where actual computing takes place. Every component will have a list of input and output ports which can be wired together and should be type checked for compatibility.

3.4 Documentation

One of the keys to survival for software written in the university research arena is good documentation. There is obviously a high turnover of personnel (students) and
thus, for a project to have any continuity, the software must be documented at least well enough for the next student assigned to the project to be able to take it over without having to start from scratch.

What often happens is that a programmer, in a hurry to try to get a demo working, will not document the code thinking that when done s/he will go back and comment it all. This rarely happens. Hence many functions are known only to the original programmer, and there is no way to find them out short of digging through the code.

What would make it as easy as possible for the programmer to effectively comment the code? One help is automatic documentation generators. There are many software development tools which facilitate the documentation of code. By using a standard format for comments in the code, an application can process the text of the code and produce a document or web page presenting the available functions along with the programmers comments in a logical way. This should clearly be standard operating procedure in any large software project. The Java language has such a tool called javadoc which can automatically document code in the form of web pages.

3.5 Network accessibility

3.5.1 Should the system be web accessible?

What benefits can be gained by making a robotic system network accessible? This question came up several times from several sources during the evolution of this project. For an experimental system, one might think that the programming overhead to make a system web accesible could not be justified- especially since it would also consume considerable cpu time during operation. However, CPUs have become fast and cheap, so eating up processing power is not a huge problem. Perhaps the bottom line is that now there is really no reason not to provide for network access (barring security considerations). While the tradeoff is a real one, a simple answer might be: "if it is running too slow, add another CPU."

One typical problem with programming systems is that it can be difficult to mon-
itor what is going on in there. While debuggers can help you step through the code, this is not really helpful in a real-time system that has many inputs and outputs. For example, suppose one writes a piece of code to find the home position for an arm by finding the joint stops of each joint in turn. When the code is run, the arm freaks out... This can be even more of a problem when the code is running on a processor with a tight servo loop and no time to spare to output debugging information. This design solves this problem by requiring complete object state to be broadcasted each time there is a significant change. In this way, the system is built with the debugging overhead built in, and hence, saving it to disk or monitoring it from another computer becomes trivial.

Another issue is that of toggling or fiddleing with parameters. This is usually very hard to do in real time. However, by allowing access to all objects, the user is not restricted to only the monitoring of their state, but may change their state or parameters in real time.

There is also some precedent for web accessible robotic systems as shown in the survey chapter.

The programming time to implement such a system could be done in such a way as to abstract away most of the network issues making programming of the system quick and straight-forward. Once the work was done, it would not have to be repeated for each new module. Hopefully it would begin to set a standard that network accessible systems in the future could build upon.

NASA/JPL has shown interest in making their experimental robotics network accessible by providing some net access to their Rocky7 prototype. This can provide scientists with the opportunity to perform science with the robot without having to travel to JPL and sit down at their control station. It makes little sense in this day of the GUI to create complex specialized hardware controls (ie. control panels layed out with buttons and monitors) when all this can be done with a well-planned GUI and a mouse, joystick, 3d mouse, spaceball, head-tracker, or any other of a myriad of standard input devices.

In addition to the obvious science benefits of providing world access to science
robotics, making the system web-accessible also seemed to be a good move in the area of public relations. Allowing citizens to log on and experience some aspects of the rover will hopefully increase public awareness and support for space exploration and science in general.

3.5.2 Possible implementations

Assuming that web accessibility is a desireable thing, what are the different ways it can be achieved and which will be most successful in helping to achieve some of the other goals stated for the system?

This project, as almost every design project, has been an evolution. Starting with a few vague ideas of how the system should be, the system specification progressed through several stages of network accessibility. Actually, network accessibility was almost an afterthought, but after many design - implementation - evaluation - redesign cycles, the possible benefits became clear and it became an integral part of the system.

It was hoped that the specification that came out of this project would be general enough to be applied to other systems. This follows from the idea that it should be possible to abstract away the majority of the networking/ distributed computing overhead, and provide a simple interface to the network and the power of distributed computing. Network computing is still in its infancy, but the infancy is nearly over as international standards are being set, paving the way to a world-wide open computing environment. Following are several possibilities for taking advantage of this world-wide environment at different levels.

Simple Socket Connection

In one of the early incarnations of the RTB system, the control code was written in C++. It implemented a socket connection to which a user interface running on another computer could connect. The user interface could then send simple text-string commands to the system to get it to do things such as move forward, turn right, lower arm, etc. While this type of arrangement would seem at first glance to satisfy much of what was wanted in terms of network accessibility, namely allowing people
to interact with the system from anywhere in the world, it presents inconveniences which became apparent while using the system.

The first inconvenience is that in this simple message passing system, it is necessary to change the code of the server running the system everytime one wants to add a new command. This involves stopping the server, adding the appropriate code, compiling, and restarting. In many cases, this requires the person wishing to add the functionality to have physical access to the computer running the code. This would make it difficult to expand the system remotely. In addition, due to the fact that the server has to be aware of every possible message and how to handle each, as the number of different messages gets large, the code can get large, messy, and unmanageable. This clearly violates some of the desired characteristics of the system, specifically, easy maintainence and extensibility.

The second inconvenience is using this type of system, it is impossible to interact with the system at the level needed for progress to take place. With a message passing system, it is difficult to provide coherent access to anything but the high levels of control on the system. However, it is often very desireable to go in and change a gain in the control law for one of the wheels on the robot, or to slightly modify some parameters the vision system is using to detect interesting objects. Accounting for all these possibilities ahead of time and creating a workable messaging system to handle all of them is a very difficult proposition. A simple message passing system does not have the generality desired.

**Remote Interfaces for each object**

When it became apparent that writing one message server to handle all the communication from the computer running the hardware to the outside world was not an optimal solution, a new approach was taken. This new approach was to create a system in which each code object provided for its own network interface. This eliminated the problem of having to maintain one “all-knowing” server, and each computing object could provide as many options to the remote computer as necessary. This would make it possible to get at all levels of the system. Dr. Salisbury advised on occasion
that “No matter how intelligent your system is, the scientists or engineers are going to want to be able to get in there and command a finger to move, or to twiddle bits.” It seemed that as long as each object provided a fairly comprehensive interface, this requirement could be met.

This was done in the Java programming language using RMI (Remote Method Invocation), which provided a fairly straightforward way to define a remote interface for each object. The remote interface would define a set of methods that could be invoked over the network. This provided a way to implement all the network functionality desired on every level.

A problem started to become apparent as several objects were implemented using this strategy. When routing all messages through a single server, it was possible to do some arbitration between inputs. The issue here is that when a robotic system is web-accessible, the possibility of receiving conflicting commands is very high. One user or “behavior” running on a computer may want the robot to move forward, while another wants it to move backwards. This problem could be solved in the single server model by simply assigning priorities to each user or behavior (based on some rule or table) when the socket connection is made. Then control of the system would be given to the connection with the highest priority, and would not be given to another until yeilded or until the connection was dropped. In the model where every object implemented its own web interface, every object and method had to take care of all that arbitration by itself. It began to appear as if the overhead required for allowing such complete web access to every object was excessive. Was there any way to abstract all that overhead and implementation detail away so that when programming, one could concentrate on making a robot system that used the web as a tool rather than making a web system that incidentally controlled a robot?

The Mirror Paradigm

An idea for a new paradigm for the system was borrowed from the web and several FTP and HTTP sites that had download links annotated with something like, “if download is too slow here try our mirror site at XXXX.” A mirror site is a site
The ideal is that the programmer will be able to create net accessible objects without having to worry about the details of implementation.

Figure 3-2: An illustration of the Mirror Paradigm.

hosted on a different computer that ideally provides a mirror copy of the original site, and with which a user can interact directly as if it were the original site. Why not provide the same type of service for individual objects rather than just entire web sites? The idea is this: Create a way to have an object instantiated on run machine, and then interact with it transparently on another machine just as if it was running locally. This turns out to be a powerful paradigm because it potentially (and ideally) allows the interaction with systems to be transparent to where the system is actually running. For instance, if in a robotic system there was a JointController object instantiated to control a specific joint of the robot, a mirror of that object could be brought up on any computer in the world with network access, and allow the user (or program) to interact with that object as if it were running on the local computer. The user would be able to change controller gains, modify simulated time delays, etc. The changes made in the mirror would be reflected back to the original, and visa-versa.

Of course there is still the issue of arbitration of conflicting commands. In the last implementation, the trouble of deciding what access to allow or disallow became burdensome. How is this system any different from that one? The difference is
that because we are establishing a programming interface which will provide all of the mirror services, that same programming interface can handle access to all of the objects which subscribe to it's service. In the first implementation, a strategy of allowing one owner of an object at any one time was employed for simplicity. While all can monitor changes in that object (the mirror is updated transparently as changes are made in the original) only the current “owner” of the object can input values into it or modify it in any way.

There is still one major issue to deal with concerning implementation of this kind of interface. How can it be done? The desire is to implement objects without having to worry about creating a remote interface and yet have remote access to all public (and protected) methods defined in the object. This is where Java comes to the rescue and provides a service not currently available in other programming languages. Java has the ability to “reflect” on a class and report back with all the methods that the class, or an object, of that class has to offer. In addition, it has the ability to invoke those methods during runtime. In general, Java allows one to get an unknown (to the programmer) object, query it to find out what methods it provides, and then invoke any of those methods as desired. In the mirror paradigm, when working with an object, if it is a mirror, it can simply package up the method invocation intended for the local object in a string format and pass it to the original object. The original object can then invoke the method and pass any return values back to the mirror from which it was invoked. Any changes in the state of the original object would get passed on to all mirrors using an updateState method. In this way, interaction is ideally transparent to location.

In reality, there will be issues with network bandwidth. If the updates are being sent too fast or if methods are being executed from a mirror at a high rate, it could bog down the network and some latency will be experienced. This is especially true when dealing with data such as images.
3.5.3 Design Requirement

To summarize the design requirement for network accessibility, network access should be allowed transparently to every level of the system. One should be able to interact with system components as if they were running on the local computer in as much as that is possible. Of course there are issues to deal with here such as network latency, access problems, security, network failure, etc., but these issues can best be explored experimentally once an initial system is implemented.

3.6 Robust error handling and Debuggability

This issue is obviously very important if one is to create a reliable system. The problem is that in large complex systems, it is nearly impossible to write bug-free code no matter how good the programmer is. However, even if the programmer were perfect, there would still be errors to deal with. This is because in any project where the computer deals with external hardware, (eg. disk drives, networks, framegrabbers, robots), there will be hardware errors and things will not always work correctly. For example, suppose that the system is running, the robot is doing interesting behaviors, and suddenly a local network server dies. Not only is this possible, it seems to happen fairly often. When this does happen, will the system be programmed in such a way as to detect the problem and act accordingly or will it go on its merry way with unpredictable results.

Another example is the possibility of a student forgetting to power the current amplifiers for the motors. When the robot tries to move nothing will happen, so the robot “tries” harder and harder increasing the value of the desired current until it hits some limit. The student seeing that nothing is happening realizes that he she has forgotten to turn on the amplifier and flips the power switch with disastrous results. The amplifiers kick on at their maximum current output, and the robot jerks wildly and smashes the joints against their limits. Yes, the student should have been more careful, and should have known to reset the system before turning on the amps. However, these are the mundane details that computers are so good at handling. So
why not use the computer to guard against such problems? The answer is probably a combination of things. First of all, error checking for every possibly problem is cumbersome and clutters code, often making the original code undecipherable. As it tends to slow down the execution of the code as well, it is often left out.

### 3.6.1 Possible Solutions

While it is impossible to completely “solve” the problem of bugs and errors, there are several things which can be done to make the system’s handling of them much more robust. Some of these are listed below:

1. Use exceptions to handle all types of errors.

2. Require all subsystems to handle all their own errors, ie. keep error handling code local to the subsystems.

3. Have a last stage to catch all unhandled errors and eliminate offending objects.

4. Have status monitors to check that mission critical subsystems are operating properly and if one is lost, try to restart it automatically as well as sending alerts to the human operator.

The use of exceptions is something that greatly simplifies the inclusion of error handling in computer code. Java has a well defined exception handling system which makes it ideal as a language for programs that require extensive error checking. In a nutshell, if during the course of running, a program detects an error of some type that it cannot handle itself, it will “throw” an exception. Throwing an exception creates a new exception object with as much detail as it can supply about the cause of the error. This stops execution of the current method on the stack and passes the exception to the calling method. If the calling method does not know how to handle the exception, then it will be passed to the next method on the stack, and on and on until a method that knows how to deal with the error “catches” it. A complete discussion of exceptions is out of the scope of this paper, but many excellent references can be found on the topic. Using exceptions automatically takes care of some of the
issues presented earlier. For one, they do not appreciably slow down the execution of the code since the exception code is only run once an error is detected. Since the passing of exceptions down the stack is handled automatically, the programmer need not constantly check return values to see if functions produced errors.

Requiring that all subsystems handle their own errors may seem obvious. But, because it is often easier to throw an exception and let someone down the line deal with it, error handling is often not done that way. For example, if a mirror is trying to contact its parent mirror, and the network is not responding, the mirror could simply give up and throw an exception. A better solution might be to inform the child mirror that the parent mirror seems to be temporarily unavailable, allowing the child mirror to make contact again after some wait period. While requiring local handling of error conditions should eliminate many problems, there will still be those that slip through and will have to be dealt with, such as programming bugs which cause low-level exceptions to be thrown.

To deal with these problems safety nets in the lowest level execution of the subsystems need to be put in place. This will protect against things such as a bug or error in one subsystem bringing down the entire computer the subsystem is running on. To create such a safety net, the method which individually invokes all the subsystems must be able to catch every kind of exception. Luckily this is fairly easy to implement in the Java exception system because you can specify a command to catch all exception types which are all subclasses of the Exception class. Once an exception has been caught at this level there are not many options. The easiest and most direct solution is to remove the offending object from execution and record it in an error log. This is the method that will most likely be used here as a first pass. A slightly “nicer” solution is to inform the offending object that it is about to be removed so that it can clean itself up and possibly invoke a new instance of itself before it disappears. The error log will record the problem with all the information that was supplied to facilitate the debugging. A stack trace is performed when the exception is thrown, which identifies the exact line of code where the error occurred.
The final tier in the strategy for robustness could be to create behaviors whose sole job it is to monitor other systems and behaviors to make sure they are functioning correctly. This is fairly easy to do since the system lends itself quite naturally to monitoring by external computers through the mirror system. When a problem is detected it could do anything from take over the function of the dead object (this is an example of computational redundancy) or try to instantiate a new object to take the dead object’s place. It could also change the mode of other subsystems causing them to pause while it tries to deal with the problem. In severe cases it would try to notify a human operator by beeping to get his/her attention or through email, etc.

3.6.2 Design Requirements

For the first stage of the system, only the first three of the above listed strategies will be implemented. The fourth can be added incrementally to the system in the future as necessary.

3.7 Summary of Design Requirements

1. Processing: The system should be programmed primarily with platform independent code; processors or computers should be connected using standard ethernet using some standard socket protocol.

2. Computing paradigm: The design requirement is a scaleable component based system. Every component will have a list of input and output ports which can be wired together and should be type checked for compatibility.

3. Documentation: Make extensive use of the automatic documentation tools that Java provides.

4. Network Accessibility: Network access should be allowed transparently to every level of the system. One should be able to interact with system components as if they were running on the local computer in as much as that is possible.
5. Robust Error Handling and Debuggability: The system will handle all errors using the exception system. Subsystems will handle exceptions locally, and unhandled exceptions will cause the generation of an error log and the removal of the offending object.
Chapter 4

ASCE Implementation

4.1 Overview

This chapter presents the interfaces and subsystems implemented to meet the design requirements presented in the previous chapter. Each subsystem is presented in turn, followed by an integration section presenting how they might all work together as a complete system. But first, a brief discussion of why Java was chosen as the primary language in which to implement the system.

4.1.1 Why Java?

To my knowledge, this project is the first to use Java to implement a complete robot control framework. While many projects have used Java as a front end for the user interface, none have ventured to use Java to write the actual control code for their robots. There are several good reasons for this. First of all, Java, being an interpreted language tends to be much slower than C or even C++. Second, it has no real-time support such as setting interrupts or scheduling thread execution. Since robots need to operate in real time, the performance hit generally cannot be justified. In addition, until recently, Java has been viewed mainly as a language only good for small applets and GUIs rather than full-blown applications.

For this Project the problems of being slow and not having real time support have
not been large issues for two reasons. First, the robot the system is being designed for has a base with two wheels, and a non-backdriveable arm. This makes the robot basically a static system so there is no need for high performance controllers needed in other robotic systems that walk or catch or perform other dynamic tasks. Second, the strategy of the RTB system is “think - small move - think - small move,” hence speed is not a critical issue. This by no means precludes the use of this system for faster robots because Java provides a very convenient way to link in “native code” (compiled code for a specific machine). Thus time-critical subsystems can be easily implemented in the language of choice on the platform the hardware is attached to. In addition, Java is getting faster and is widely accepted (it was recently voted to make Java an international standard). There are even compilers appearing to convert the Java into much faster machine code to run on specific machines. The speed issues are really minor when compared to the benefits which Java affords.

Briefly, Java has the following features which are heavily used in the implementation of the ASCE system:

1. Reflection - Java allows for the runtime system to find out the structure of any class object (its variables and methods) and then use that information to invoke methods or access its variables. This functionality is essential to the operation of the mirror subsystem described in this chapter.

2. Remote Method Invocation - This allows the invocation of a method on a remote computer similar to the Remote Procedure Call (RPC) system. This is also essential to the mirror subsystem.

3. Java Native Interface - The Java Native interface provides a very clean and easy way to connect Java code to machine specific native code. This is necessary for interacting with device drivers for things such as frame grabbers and joint servos.

4. Abstract Window Toolkit - This provides a convenient and easy way to create user interfaces which integrate naturally with the rest of the system rather than requiring large amounts of glue code as is necessary when using something like TCL/TK to create the user interfaces.
Applets - They provide a convenient way to run java code over the web. It requires a java-compliant browser which will run the java program in its memory space.

### 4.2 The Mirror Subsystem

The general idea of an object mirror was presented in the last chapter, here the actual implementation will be presented. The goal in general was to facilitate revisions without having to revamp the entire system by keeping the subsystems fairly independent of each other. While this has been somewhat difficult, due to this subsystems central nature to the whole framework, it has been achieved in part by deciding that all network issues were to be handled by the mirrors. On a practical level, that means that no other part of the system needs to know anything about the network, and can operate as if all components with which it is interacting are running on the local machine. This abstraction takes away any concern about how to create a connection between machines, because the connection has been taken care of by the mirror.

In the following discussion some definitions would be useful:

1. **mirror**: Any object or class which implements asce.mirror.Mirrored and uses a Mirror object as a helper.

2. **root**: The original object, topmost object in the mirror tree, a mirror with no parent.

3. **child**: Any mirror which has a parent.

4. **parent**: Any mirror with a child.

5. **leaf**: Any mirror with no children.

#### 4.2.1 The mirror model

There is no distinction in this model as far as the classes go, between being the root object and child. When programming a new class one must provide for operation in
Figure 4-1: Shown here is a possible mirror tree. Notice that there is no conflict with having two mirrors for the same root running on the same computer as long as they have unique names. In general this is not a very efficient way to do things, however.

all modes: root, parent, child, leaf. This is because all children objects are potential parents. It was decided to do things this way to leave open the possibility of multi-level trees of mirrors, (See Figure 4-1). In practice, there will usually be only two or three levels. One reason this capability is desireable is to allow the programmer to protect the original object from having to keep hoards of mirrors up to date while also trying to complete its primary goals, whatever they may be. For instance, the root BaseController is a fairly critical object. It needs to output new currents to the motors on a regular basis. If it was trying to keep several mirrors up to date, it wouldn’t do a very good job of controlling the base. So instead, we can set up only one child for the root on another computer, and then all other children can mirror the first child. In this way, it is possible to insulate critical components from the load of servicing many mirrors. (See figure 4-2 for a better arrangement) Note that if a mirrored object is set up correctly it will provide all submirrors with a complete picture of its state at regular intervals for a rapidly changing object, or whenever a
significant change occurs for objects which change more slowly. This will facilitate debugging, as the user will be able to monitor the state of any object in the system, and can easily implement elements that will save the state to disk for later analysis. The mirror system will also make object interaction by both humans and processes simple, opening the doors to a very powerful distributed system.

4.2.2 How mirrors work

The asce.mirror.Mirror class can be thought of as a helper class which takes care of most of the work associated with maintaining a mirror. It makes heavy use of two java subsytems to accomplish it’s work, the java.rmi and java.lang.reflect packages. Java.rmi is a package which provides functionality for Remote-Method-Invocation. It allows the programmer to define a remote interface for an object, and then invoke the methods of that object from a remote machine. In the early stages of the project, a remote interface was being created for every object in the system. This quickly became very tedious and allowed thought such as, “no one will ever want
to mirror this class, so why bother writing the interface?” This threatened the goal of having every computing object mirrored. A new strategy was necessary. After some brainstorming it became apparent that by using the java.lang.reflect class in conjuntion with a Mirror, that the need for creating a remote interface for every mirrored object could be avoided. Reflection allows the java to find and invoke object methods during runtime. Using this ability, it became possible to create asce.mirror.Mirror:executeParentMethod(String cast, String method, Object[] args) method. This allows the mirror to pass a method call through to the parent by specifying the class-type and the method name as strings. The parent mirror will then find and invoke the correct method in the mirrored object. This approach requires that any arguments passed to the method be serializable (see Appendix D.2 to find more information on serialization). However, this was a requirement for the arguments of every method in a remote interface, hence there is no additional inconvenience here.

On the surface it might seem that this sneaky approach to creating remote objects has all pros and no cons. After all, it provides a way to remotely invoke any method in the parent object without having to explicitly create a remote interface for every such object. It also has the ability to keep the child’s state up-to-date using the outputState method. These two features make it very convenient, even trivial to create mirrored objects, objects that (minus possible network latency) completely insulate the user from the issue of where those objects are actually running. However, there are actually two cons to this convenience. First, there is an issue of speed. Using reflection to find and run methods from string names is not nearly as efficient as calling them directly using RMI. Here, as is typical with the whole framework, convenience and ease of use wins out over absolute speed. The second con is that having a local copy of every mirrored object can start to eat up memory when the system grows large. However, given that a single computer will probably only need to open a portion of the available mirrors at one time, this should never be a serious issue.

In an earlier version of the mirror package, it was required that a class subclass the asce.mirror.Mirror class in order to be mirrored. This was seen as restrictive and is no longer the case. A class merely needs to have a Mirror object, and implement
the asce.mirror.Mirrored interface. The Mirror object provides methods to its Mirrored object for registration in the local computer's object registry, connection to its parent's Mirror object (which allows the parent to keep the child's state current), and execution of the parent's methods. Anytime the parent object transmits new state data, the child object is notified using its mirrorStateChange method. The mirrored object has the responsibility of updating the local state with the newly transmitted state, as well as passing any method calls that would change the object's state up the mirror tree to the root using executeParentMethod. One might ask why it is required to pass the method calls that might change the object's state up the tree from child to parent rather than directly to the root. One example of why this is necessary is (as will be seen in the arbitration section), because sometimes method calls can be filtered out or handled by objects before they get to the root. Appendix B.0.1 has a discussion of the mechanics of implementing a mirrored class.

When a new object mirror is created, it starts out as a root by default. However, it is not available to be the parent of other mirrors until it is registered in the rmiregistry. The rmiregistry is basically a named java-object server that needs to be running on each machine that will be host to mirrored objects. It provides methods for registering objects with a String name so that other computers can request those objects by name. (For more information on the rmiregistry see Appendix D.2.) The mirror class takes care of registering the mirror in the registry when the mirrored object calls asce.mirror.Mirror:bind(mirrorName) method. Once the mirrors bind method has been called, the mirror is available to be a parent. Note that it is not necessary that every mirror bind itself in the registry. Not doing so will only preclude the possibility of having children mirrors. At this point the mirror is, by default, a root because no parent has been specified. However, at any time the mirrored object can call its mirror's asce.mirror.Mirror:connectToParent(parentName) method which will look up the parent mirror and register itself as a child. This will enable the parent to output state updates to the child, and will provide the child with a parent on which to invoke methods when desired. Note that the parentName string passed to the connectToParent method needs to have the following form: //completehost-
name/objectname, eg. //europa.ai.mit.edu/SimpleBaseController, so that the system will know on which computer to look for the mirror.

**Mirror Operation Example**

This subsection will present an example of a mirror’s creation and operation. This example will not include arbitration, as that subsystem has not been presented yet. In this example, a mirror will be created to monitor the frame-grabs made by the vision system. Suppose that the frame-grabber for the robot was on a computer called rtbeyes.ai.mit.edu. When the programmer creates the new frameGrabber class, s/he will create it as a Mirrored object which includes an instance of Mirror in its member variables. The class will be written so that when it is instantiated, it will check to see if it is running on a computer with a frame grabber. If so, it will connect to that hardware, initialize its state (by making an initial grab) and register itself in the rmiregistry under the name FrameGrabber. It then sits and waits until a child mirror is created. Now, a user on starfleet.umd.edu wants to monitor what the rover is looking at. S/he can do this by instantiating a FrameGrabberGUI class. This is a class which subclasses FrameGrabber, and adds a graphical user interface to it to display the image and present the user with options.

When the FrameGrabberGUI class is instantiated, it checks on the local computer to see if it has frame grabber hardware. Finding none, it assumes that it will be a child mirror, and prompts the user for the name of the parent (or looks it up in a config file). The FrameGrabberGUI object will then use the rmiregistry running on rtbeyes.ai.mit.edu to look up the FrameGrabber object running on that machine. The registry returns the object reference and the FrameGrabberGUI object then calls its register method. Upon registration, the parent sends the current state to the child (which includes the latest frame grab). The FrameGrabberGUI object can then open its GUI and present the image as well as some buttons to perform various actions. The user then decides that s/he wants to switch the camera channel. S/he presses the channel 3 button on the GUI. The GUI calls its superclass’s setChannel(int channel) method. In that method, the frame grabber mirror checks a variable called “local”
to see if it is a root or a child mirror. Upon finding out that it is a child, it uses
the executeParentMethod("FrameGrabber","setChannel",1) to pass the method call
to its parent. The child mirror then takes the arguments in that call and transmits
them over the internet to the parent mirror using the executeMethod call. The
parent mirror receives the call, uses java reflection to find the method “setChannel”
in its mirrored object (FrameGrabber), and then invokes that method. The method
in FrameGrabber then checks to see if it is a root. Finding that it is a root, it
tells the hardware to change the channel and then take a new frame grab. When
the new frame grab is complete, it realizes that its state has changed so it calls
the mirrors outputState(state) method. This method passes the new state to all
registered children mirrors. The child running on starfleet.umd.edu receives the new
state (which includes the new image) and updates its GUI to present the image to
the user.

For a better understanding of how mirrors work, one should read Appendix B.0.1
on how to program using mirrors, and try to create simple mirrored classes. While
the previous example may seem confusing, implementing mirrors is actually quite
straight forward.

4.3 The Arbitration Subsystem

As described in the previous chapter, many revisions have produced the current arbi-
tration subsystem which is described here. Yet no claim can be made that it is without
flaw. In fact, there are several that will be explained in section 4.3.3. However, this
implementation was eventually chosen due to its simplicity both in programming and
in operation.

4.3.1 The Arbitration Model

This implementation of the arbitration subsystem resides in the asce.arbitration pack-
age. It contains the following:

1. Arbitrated interface
2. Arbitrator interface

3. Key class

4. DefaultArbitrator class

This system is similar to the mirror system in that objects which are to be arbitrated will implement an interface and use a helping class. The arbitrator interface defines several methods that need to be implemented by any arbitrator class.

4.3.2 How Arbitration Works

Each asce.arbitration.Arbitrated object has at least one asce.arbitration.Arbitrator which helps it decide if a particular method call should be allowed. The Arbitrated interface has only one method: getArbitrator. Using this method any object user who wishes to have access to an arbitrated object’s methods can request the arbitrator associated with it. The object user can then check if it’s key has sufficient priority to gain control of the object by calling the arbitrators canTake method. If the object user wants to take control, it can submit its key to the arbitrator using the addKey method.

All arbitrated methods require a Key as the first argument in the method call. One of the first orders of business in the body of the method is to see if this method was invoked by an object with the proper key. It does this by invoking the asce.arbitration.Arbitrator:checkKey(key) method which will throw an InvalidKeyException if the key is not valid. The exception is not caught, but is allowed to return to the invoking object to inform it that it does not, or no longer has the correct key. In the current implementation (asce.arbitration.DefaultArbitrator), each Arbitrator can have only one correct Key at a time. An object wishing to use the Arbitrated object can submit its key to the Arbitrated object’s Arbitrator using the addKey(key) method. This will put the key in an ordered list and the key with the highest priority in the list will become the correct key.
An Arbitration Example

Figure 4-3 presents a simple graphical representation of Arbitration. The first object "fooism" is an arbitrated object with a method that object "fooismusr" wants to invoke. In the top half of the figure, the state of the objects is shown as the first line of code in the window of the fooismusr object is being executed. This line of code "arb = fooism.getArbitrator();" gets a reference to the fooism’s arbitrator (hence the dashed arrow pointing to the arbitrator. In the bottom half of the figure is shown the state of the objects soon after the second line of code in the window is executed. This line of code adds fooismusr’s key to fooism’s arbitrator. This example assumes that no other keys had previously been submitted to the arbitrator. Therefore as soon as fooismusr’s key is submitted it becomes the highest key. In the next step, which is not shown, fooismusr will invoke fooism’s foo method with it’s key as the first argument. The foo method first checks the key using the arbitrator, and if no exception is thrown, it will execute normally.

If the arbitrator had other keys submitted previously, it would reorder its list upon the submission of the new key, putting the key with the highest priority on top. In the case where fooismusr’s key was not the highest, if it tried to invoke the foo method, it would throw an Invalid Key Exception.

4.3.3 Shortcomings of this implementation

There are several shortcomings of the current implementation of the Arbitration system. While implementations of several similiar arbitrations schemes with more functionality where tested, this “simplified” version was eventually chosen to facilitate integration with the rest of the system. The first and perhaps most troublesome limitation of this implementation is the possibility of dead-lock.

Deadlock

Due to the fact that a behavior may need control of several Arbitrated objects in order to do its job, it is very probable that not all of those objects will be available
object fooism
implements Arbitrated

myArbitrator:

method foo(Key k, ...)

public void foo(Key k, ...){
    myArbitrator.checkKey(k);
    do lots of neat stuff...
}

Figure 4-3: An arbitration example.
at the same time. Perhaps a higher-priority behavior will have control of one or more of the objects needed. If an object uses the arbitration system “nicely” it will wait to take control of any needed object until it knows it can take control of all needed objects. (Remember that taking control is accomplished by submitting it’s key, and if it’s key is higher than all the rest of the keys in the list it will be granted control.) A behavior can check that it’s key has sufficient priority to take control of an object without actually submitting it by calling the asce.arbitration.Arbitrator:canTake(key) method. However, a poorly written object may take control of some of the objects it needs, hence tying them up, while it waits for the other objects it needs to become available. This could easily result in a system dead-lock where everybody is waiting and nobody is working. Hence in this implementation of the arbitration system the responsibility for preventing dead-lock is relegated to the objects using the system. Earlier implementations handled all of this by stipulating that an object only had to specify which Arbitrated objects it needed. The arbitration system would then notify the object when it had control of all needed resources. This system is preferable, and future revisions should move towards it.

Security

Another current limitation is that this system currently provides no real security against unauthorized use. This is because the asce.arbitration.Key class is acting as a place-holder for a more comprehensive and secure Key class. Currently the Key class consists of a 64 bit random signature used to identify it and an integer specifying its priority, with no encryption used. This is why the current version is not made available in Applet form. Any user must have a complete copy of the Java code on the local machine before they can interact with the system in any way. Even then, a hacker could intercept packets and send commands purely on the TCP/IP level. Clearly, before this type of system is used to control sensitive hardware or information, a much tighter security system will have to be put in place.
4.4 The Port Subsystem

One of the original goals for the RTB system was to use it as a platform on which a scripting language for remote grasping tasks could be developed. Most scripting languages are currently based on a linear procedure-oriented programming paradigm. This is how several exploratory attempts at defining a scripting language for this system developed as well. However it was soon noticed that certain aspects of this approach did not fit well with the problem of controlling a robot. The linear-programming approach is one of “do A then B then C if D do E else F etc., often passing in arguments and getting return values.” The robot control problem is often better described in terms of data-flow.

Visual component-based programming has become popular in the past few years due to the success of object oriented languages such as C++, and so why not create a scripting language based on that paradigm rather than the old paradigm of linear programming? The desire was to create a system where the scriptor could specify connections between the output ports and the input ports of objects. If a complete system could be built up by connecting output ports to input ports in some type of visual editor, it would also lend itself quite easily to scripting. For example, a SimpleBaseController has been implemented which uses input and output ports of two subcomponents: BaseEncoders, and BaseMotors. The BaseEncoders component has two output ports: “Time”, and “EncoderValues” which the SimpleBaseController connects to it’s internal “Time” and “Pos” input ports. This is done by first getting the list of output ports offered by BaseEncoders by calling getOutputPorts(). It then looks through the ports to find the one with the right name, and when the time is right, it can connect the ports together.

This implementation of the port subsystem resides in the asce.port package. It contains the following:

1. InputPort interface

2. OutputPort interface
4.4.1 How Ports Work

Like the other subsystems, the port subsystem has gone through many iterations to reach its current state. It has been revised many times in an effort to get it to integrate easily and naturally with the mirror and arbitration systems. The OutputPort interface has only one method, getData(), which returns an Object. This method should only be called by an InputPort connected to the output port. An InputPort has a connect(outputPort) method and a disconnect(outputPort) method. To connect an input port to an output port, one simply gets references to the input and output ports, and then passes the output port to the inputPort connect method. It is completely up to the object implementing the InputPort as to how handle other issues. There is no specification as to whether InputPorts or OutputPorts can handle multiple connections, however, each OutputPort can be connected to several InputPorts, but each InputPort can be connected to only one OutputPort.

A PortOwner is required to implement two methods, getInputPorts, and getOutputPorts. Each method returns an array. This is the only way for other objects to gain access to an object’s port. Note that an object does not need to return every port it has, as some might be soley for internal use. These can be thought of as private or internal ports, while those returned in a call to get would be public or external ports. Figure 4-4 shows an example of external ports, (those shown on the BaseEncoders and BaseMotors objects), and internal ports, (those shown in the SimpleBaseController object).

In addition to the methods listed above, the InputPort interface also has a method called testDataCompatibility(Object data). This method throws a PortException if the port is not compatible with the data passed in. This is intended for use by a visual application builder which will allow the user to connect ports graphically by clicking
Figure 4-4: An example of external or public ports vs. internal or private ports. The BaseEncoders and BaseMotors objects are shown inside the SimpleBaseController because it abstracts them.
on an output port and dragging a connection to an input port. A first generation visual application builder has been programmed, however it was intended more as a proof of concept than a full-fledged application. Hence, it will need significant work before it can be used as such.

4.4.2 Issues in Process Flow

Once a network of objects are connected using the input and output ports, how is the actual processing accomplished? There are several possibilities here. One of the first approaches taken was to have every high-level behavior running with its own thread. That behavior could handle the execution of the lower level components it used. This system would work fine if there was more control over the handling of threads and more consistency among various operating systems in how they handle threads. In the end, it was decided that the overhead and uncertainty were too big a price to pay.

The next approach attempted was to create a ServoLoop object which was responsible for the execution of each “Servoable” object in the system. A Servoable object had a no-arg method named “calc” that would be invoked each time the servoloop ran. This worked fairly well, but a couple of other methods were tried for comparison sake.

One idea was to try and let the dataflow force the processing. There are two ways to do this, a “data push” and a “data pull.” In the data push method, there would be data sources that would start the processing by pushing their new data to the input ports connected to them. Each time an input port received new data, it would do the appropriate calculations, then pass the results to its output port. The output port would then push it to the input ports connected to it, and on through the system. While this method seems fairly natural, there are some complications inherit in it. The first is that in an object that requires data from several input ports in order to do its calculations, there is a timing problem. There would have to be some mechanism devised so that processing would wait until all the required input ports had been filled with new data.

In the data pull method, this problem disappears. Each input port would request data from the output port it was connected to when needed and this would force the
processing back through the system. Since we currently only allow one output port to be connected to each input port, this would be fairly straightforward. For instance, suppose the amplifier object for the base motors was set up to request new current values approximately every 10ms. It would query its current input ports which would query the output ports connected to them and force processing in that object (in this case the BaseController), and on through the system. Data pull certainly seems like the way to go, however, it was found that it required the surrender of control of the execution of individual elements to the data passing through the network of ports. This created the possibility of a thread hanging, perhaps getting stuck in some loop, or throwing an exception that could stop the whole system. So in the end the ServoLoop method was chosen and several improvements were made to the first implementation.

4.5 The ServoLoop Subsystem

One of the main goals for this framework was to have a robust, error-tolerant system. Meeting this goal is one of the main jobs of this subsystem. This is why the ServoLoop system won out over other processing systems explored. The main idea is that any object which needs periodic processing can implement the asce.servoloop.Servoable interface, which has just one method, “calc”, and then be added to the list of objects to be run in the servoloop. At first all objects on the list were run every time through the servoloop. It became clear that this was unnecessary as different objects needed different servo rates. At first, an attempt was made to allow each servoed object to specify the rate at which its calc method would be called. This was found to be impractical as maintaining the ordered lists of which object needed to be executed next was a time consuming operation. Instead, a more quantized approach was devised in which three different approximate rates are offered. The asce.servoloop.ServoLoopThread class provides three methods for adding objects to the servoloop: addHigh(Servoable), addMid(Servoable), and addLow(Servoable). (The high, mid, and low refer to the rate at which the object’s calc method will be called, not to be confused with high,
mid, or low-level control.)

On the several computers tested, the period of the high-speed servoloop averages anywhere from 1ms to 60ms depending on the load on the system. The mid level objects will be executed on a period approximately an order of magnitude greater than the high rate, and the low rate’s period will also be about an order of magnitude greater than the mid rate’s. This was accomplished by having the servoloop execute every high rate object followed by a thread-sleep of several milliseconds. Every tenth time, instead of executing the thread sleep it would execute one of the mid level objects. Each time the mid rate objects had all executed, one of the low rate objects would be executed. The system was set up this way to spread the processing requirements out as evenly as possible. It seems to work out quite well. Obviously, some parameter tweaking is useful to tune the performance on each system. Note that it is important that the calculations performed in each calc method be concise and return quickly.

4.5.1 Fault Tolerance

Using the ServoLoop subsystem for all the processing provides a very convenient way to isolate and deal with errors and exceptions. (For more information on errors and exceptions see Appendix D.2.) Each time a calc method is executed from the servoloop, there is the potential of it not executing correctly. However, because the servoloop has the ability to catch these exceptions and errors, it can simply remove the offending element, and generate an errorlog, then continue executing. One could easily implement a high level behavior which can monitor for errors and react accordingly.

4.5.2 Real-Time Support

As alluded to earlier, one problem with the system is that Java offers no real-time support. Hence, there is no way to guarantee that the servoloop will be executed as often as it should. This, in practice, has not been a problem for the simple fact that any computer that is running high-rate servos is dedicated solely to that purpose,
so their is not a lot of competition for the cpu. The longest wait time experience
between execution of the high-rate servoloop has been about 120ms, which is rare,
and hasn’t caused any trouble. If this system were used in an application where
the high rate servo needed more consistency, one could write system specific code
to generate interupts as needed to execute the servoloop. This option has not been
explored here.

4.6 The Behavior Subsystem

The system is built around the component paradigm presented in the port section, and
uses both Mirrors and Arbitration, hence it is an integration of the subsystems that
have been presented so far in this chapter. Behaviors can be thought of as individual
computing objects used to abstract away complexity and to create logical groupings.
Some examples might be a BaseController, a VisionSystem, etc. This is a very loose
use of the term “behavior.” Perhaps the word “component” would be more accurate,
but as the package has been named asce.behavior, we will continue to use that term.

4.6.1 How Behaviors Work

A behavior can be in one of three states at any time. Those are:

1. Dormant - The behavior has been instantiated, but has not been enabled with
   a key.

2. Enabled - The enable(Key key) method has been called. In this state the
   behavior will wait until it wants to run (depending on its function), and then
   try to gain control of all required resources.

3. Running - When the behavior has acquired all required resources it will wire
   their input and output ports appropriately, and then it will run.

   Note that every behavior must be enabled with a key. Recall from the section
   on arbitration that every key has an associated priority. It is these priorities that
determine which behavior will have access to the lower level behaviors at any one

time. With the system set up in this way, it is possible to implement several types of

robot control systems. For instance one could think of the behaviors as procedures, or

finite tasks. Then a central controller could be written which would simply invoke each

behavior in order. In this case, because there would be only one centralized controller,

there would never be more than one simultaneous request for a particular resource,

and the arbitration system would not really come into play. On the other hand, if a

more behavior-based system was desired, one could simply start each behavior with

a priority appropriate to that behavior’s function. For example, a “roam behavior”

might be started with a low priority and an “avoid cliffs behavior” would be started

with maximum priority. In this example the roam behavior would immediately take

control of the hardware and start roaming, while the “avoid cliffs behavior” would

wait in the enabled state until a cliff was sensed by a vision, sonar, etc. When a cliff

was sensed, it would then take control of the hardware from the “roam behavior”

and steer the base away from the cliff. When a safe distance away, the “avoid cliffs

behavior” would release control of the hardware, and the roam behavior would start

running again.

4.6.2 The Handling of Mirrors

The nice thing about the way the mirror system was created is that they are handled

transparently by the behavior system. The behavior manager does not need to know,

and does not care, whether a particular object is a root mirror, a child mirror, or if it

is even mirrored at all. As long as each mirrored class is set up correctly, the rest of

the system will never know that the root object is not running on the local machine.

4.6.3 The Behavior Manager

One issue that deserves some explanation is how a behavior gains access to sub

behaviors. This is taken care of by the asce.behavior.BehaviorManager class. When

the system is starting up, every newly instantiated behavior is submitted to the
BehaviorManager which maintains a list of all the behaviors in the system. When a high-level behavior needs access to another behavior, it can request a reference to that behavior by name. The behavior manager will look through the list of behaviors and return the first one that has the given name.
Chapter 5

The MIT Rover Test Bed

5.1 Overview

This chapter presents the MIT Rover Test Bed in detail and discusses the application of the ASCE system to it. For clarity of discussion, it is helpful to note that the ASCE system exists in the form of Java packages all beginning with the prefix asce. Some examples are asce.arbitration, asce.mirror, asce.servoloop, etc. Conveniently, the code for each package is found in a directory whose path corresponds to the package name. Thus the asce.arbitration package can be found in the .../asce/arbitration/ directory. The code written specifically for the rover are found in packages starting with the prefix rtb. The first part of this chapter gives an overview of the hardware, and the remainder is devoted to explaining the rtb packages.

5.2 Hardware

5.2.1 The Base

The rover base consists of a rectangular aluminum frame with approximate dimensions of 30cm wide by 20cm high by 50cm long (see Figure 5-1). There is an umbilical coming out of the back of the base which has wires for all of the motors and two thin coax cables for the video from the cameras. The base has two driven wheels in
Figure 5-1: The rover before the miniwam was mounted. The two sphere were used in a behavior that would visually servo the rover to a lander mockup using the overhead camera.

front (see Figure 5-2), and two caster wheels in back. All processing is performed offboard. As this rover was built as a testbed mainly for developing grasping algorithms, it was not necessary to create a base capable of traversing harsh terrain. The rover has a playpen made of plywood measuring approximately eight feet by eight feet, the playpen has a one foot high plywood wall around it to keep the rover from wandering off the side.

5.2.2 The Arms

Currently the rover is outfitted with two arms. The left arm is the first-generation JPL Rocky-7 arm. It is a two dof arm with an ingeniously designed two dof end-effector which has fingers that can rotate 360 degrees. One side of the fingers are used for grasping, the other side is used for scooping (see Figure 5-3). The right arm is a four d.o.f. cable driven arm called the mini-wam built by Thomas Massie as a small scale clone of the WAM robot built by Bill Townsend (see Figure 5-4). Currently Arrin Katza is building a two-dof end-effector for the mini-wam that will
Figure 5-2: A close up of the drive system.

Figure 5-3: The JPL arm.
allow coring and grasping. The first arm the robot was outfitted with was a one-dof arm with a two-dof end effector. The arm was intended to be used as a functional mockup of the JPL arm. Since one of the degrees of freedom on the JPL arm is used primarily for stowing the arm, rather than to aid in grasping, the one-dof arm was fully functional. Its end effector was designed and built by Hani Sallum, it has two cable driven fingers outfitted with force sensors. It was replaced with the JPL arm (on loan from JPL) in order to make use of the unique abilities of their end-effector.

5.2.3 Vision

The rover has a pair of cameras mounted on it to allow for stereo image processing (see Figure 5-5), although to date we have used only one camera at a time. This is possible because the rover currently operates on a flat surface, and therefore any objects in the workspace will be lying on that surface. With the surface providing one constraint, all that is needed is the input from one camera to uniquely determine the position of the object. There are also two cameras mounted over the playpen to simulate lander cameras.

The system uses the Matrox Meteor frame grabber running in a PentiumP-5 166.
The frame grabber is not high performance, but is adequate for a look-think-move strategy. It allows for up to four camera inputs, but can capture from only one line at a time. After switching the active line it takes a couple of seconds for the signal to sync up. Again, this wait time is no problem for this system since we are not attempting to carry out any time critical tasks.

5.2.4 The Brains

In addition to the computer just mentioned which is dedicated to the frame grabber, there are four other computers dedicated to this project. There are two for controlling the various motors associated with the base and arms, one for running mid and high-level behaviors, and one to act as a server for the mirrors. Figure 5-6 shows some of these computers.

The computer that is used to control the motors for the two wheels on the base, and the four joints of the JPL arm is a Dual Pentium 180. It has an ISA slot card developed by Sensable Devices that can read digital encoder output and output voltage levels to the amplifiers which are mounted in a rack next to the playpen. The wires for each motor run through the umbilical to the rover. A dual Pentium 200 is dedicated to controlling the mini-wam and uses the same setup. The computer
used for running mid and high-level behaviors is also a dual Pentium 200. Its only connection to the rest of the system is through Ethernet. It monitors the other parts of the system (the vision, the base, each arm) by creating mirrors of those objects that it is interested in. It connects to these mirrors through the last computer, an Ultra-Sparc which acts as the mirror Server. As requests for mirrors come in, if the server does not already have that mirror, it will set it up. When all connections to a mirror are dropped, it will then drop the mirror as to not tie up the network by transferring information that is not being used. Figure 5-7 should make the role of each computer clearer. Note that this is just one possible configuration. The system has been designed in a general way so that any lay-out of computers is possible.

It is important to realize that the five computers just described are computers dedicated to the system, however, every other computer on the Internet is a possible addition to the system. Any computer can connect to the mirror server (granted that it is listed as a trusted host), and interact with the objects in the system, from the lowest-level, such as an encoder value input object, to the highest level, such as a mission planner element.
5.3 The RTB System Layout

The system has been set up in such a way that only the low-level servoed objects run on the computers attached directly to the hardware. This conserves all the processing power for the running of the high-speed servo loop and for the output of system state to the mirrors. The rtb packages are broken up into functional groups, for example the rtb.motor group contains all packages that have to do with robot producing motion. The current packages are:

1. rtb.motor.motoramp - This package contains an interface (MotorsAndEncodersInterface) that defines interaction with a generic set of motors and encoders where there is one encoder for each motor. There are currently two implementations of this. There is the MotorAmpArray which interfaces with the DtoA and encoder reader ISA boards used to control the amplifiers. The other is called SimulatedMotorsAndEncoders and, as its name implies, it mathematically implements a very simple mass, damper system to simulate a motor hooked to an encoder. It is not intended to be accurate, rather, to simply implement the interface so that code can be tested without actually running the hardware.

2. rtb.motor.base - This package contains most of the code used for controlling the base. It defines BaseEncoders- which reads the encoder values from the two driven wheels, BaseMotors- which allows the setting of the current output to the two wheels, SimpleBaseController- which accepts simple joint space trajectories, and controls the base accordingly. There are also some smaller supporting classes such as exceptions. The trajectory class is defined in asce.trajectory, and allows for the creation of a trajectory consisting of times and joint angles for any number of joints. For convenience, it is quite easy to subclass the trajectory class to build the simple trajectories that will be used most often. For instance, there is a MoveForwardTrajectory and a TurnTrajectory which, in effect, are all that are needed to get the rover to any desired spot. The constructor of each takes a desired average velocity and final position, and then creates a trajectory to accomplish that motion.
Figure 5-7: A layout of the RTB System. Note that only a few components are shown, dashed round rects represent object mirrors.
3. rtb.motor.joint - This was intended to be a package with utilities for controlling the robots arms or base in joint space, but is currently not being used because this functionality has been included directly in the base package, and the arm packages have not yet been updated.

4. rtb.vision.camera - This package holds the interface and the code for interaction with the frame grabber. It also has GUI classes for displaying the frame grabs on the screen, or mirroring them to another computer.

5. rtb.vision.processing - This package holds classes for processing images obtained by the cameras. Currently there is just a blob detector implemented which can detect blobs of white, green, or blue depending on the settings.

Further information on the rtb package can be found in the online documentation. (See Appendix D.1.)
Chapter 6

Experiments

6.1 Overview

This chapter presents some sample behaviors that have been implemented using the system. Many of the tests executed where done in simulation, however, the last test presented integrates all of these and executes them on the actual rover hardware. These tests provided many insights on how to better the system in addition to validating the general approach taken. While the system currently is far from production quality, it was found to be fairly easy to use, both in programming-time and run-time.

Unfortunately, there are no easily definable, quantifyable metrics for the system. While it would be possible to collect and graph data such as response time or task completion rate, they would be meaningless without something to compare them to. Hence, the more qualitative approach will be found here. Keep in mind that the following experiments exercise all of the subsystems described in the previous chapters even though they are not mentioned explicitly. More than anything, the tests served as a system integration check, verifying that they can be made to work together.
6.2 Task Experiments

This section presents some experiments in task execution. Task execution mainly exercises the Arbitration and the Port subsystems. The main results appear at the end of this section. The beginning of the section presents some early experiments, and how their results guided the project.

6.2.1 Early Sample Return

Early in the project a set of behaviors were written that successfully controlled the robot through the execution of a basic sample return strategy. The behaviors were set up to operate in a sequential fashion, rather than running in parallel as a prioritized behavior based system. This experiment worked well when each step executed successfully, but when a failure occurred in one step, recovery for the high-level goal (returning an object to the lander) was difficult.

The behaviors

The sequential behaviors needed to carry out this task were the following:

1. Roam - in this case, since the playpen is small, the roam behavior was implemented as simply turning in place.

2. See Interesting Object - the image processing behavior was set up to find white, green, or blue blobs. In this case we decided that interesting objects would be white ping-pong balls. (Wouldn't it be neat to find them on Mars!) This behavior would notify a commanding behavior that an object has appeared in the view of the rover.

3. Servo Base on Object - this would move the base such that an object viewed in one of the rover’s cameras would move to a specified position. For example, if the object was too high on the screen (meaning too far away), the rover would move forward. If the object was too much to the left, the rover would turn left, etc.
4. Grasp Object - This would unstow the JPL arm, and grasp the object.

5. Servo Base to Lander - This used the two colored globes on the rover (shown in Figure 5-1) and the lander camera to visually servo the base to a pre-specified location near the lander.

6. Deposit Sample - This would first visually servo the base on a visual target on the lander and then would release the sample.

The sample return behavior integrated the previous behaviors to carry out the task. The rover would be placed in the middle of the playpen, and a ping-pong ball would be placed randomly. The sample return behavior would then be started. The roam behavior would start as well as the see interesting object behavior. When the interesting object behavior saw the ping-pong ball, it would stop the roam behavior. The “servo base on object” behavior would then monitor the object’s position in one camera view, and move the base to a prespecified location with respect to the ball. The servo base to lander behavior would then turn the rover towards the lander and move it into position so that the rover camera could see the lander visual target. It would then servo on that, moving to the drop off position, and open the hand to release the ball.

This behavior had a very good success rate, however, in those rare times when something went wrong at one stage, it was often necessary to start over. Due to the fact that in the linear programming approach, it was necessary to keep track of the state of the task, if something out of the ordinary happened it was difficult to determine the new state without a complex tree of if-then type statements. This was the approach taken for a similar experimental task written for the WAM system.

**WAM object retrieval**

The similar task written for the WAM system was to find and grasp objects in its workspace and deposit them in a bin. While the WAM has a fixed base, the experiment is very similar to the previous one described for the RTB, the difference being that the FEGs would “roam,” rather than the base, until an object was found. This
task is of interest because it was programmed as a decision tree, a large if-then-else type structure. While the decision tree was eventually developed to a point that the task completed successfully essentially every time, this had only come after running many tests, observing the possible failure modes at each step, and recovering from them appropriately. In the end, the amount of code supporting state verification, and error-recovery dwarfed the code for carrying out the individual sub-tasks. In addition, this approach was not found to be very robust to changes in the programming, ie making a slight change in the decision tree, or modifying a parameter could, (and often did) cause the total failure of the task.

The Behavior-Based Alternative, for Error Recovery and Opportunistic Science

This task might better lend itself to a more prioritized behavior based approach. This is especially true when trying to create a system to support opportunistic science. Opportunistic science might be best explained by an example. Suppose there is a primary goal for a rover to make a traverse to some site where it will perform some science. Along the way, it passes a very interesting mineral deposit. Without an opportunistic approach, the rover would drive right by the mineral deposit, its only goal being to get to the pre-determined site. In an opportunistic approach, the rover would be programmed to notice interesting things and take advantage of the opportunity to study them. While the opportunistic approach could be programmed using a decision tree, it lends itself very nicely to the behavior-based approach and could be accomplished in the following way. A traverse behavior would be started with a medium priority, and a find interesting deposits behavior could be started with a higher priority. The find interesting deposits behavior would just monitor the vision system passively until it saw something of interest, at which time it could exert control of the rover, collect data from the mineral deposit, and then release control, returning it to the traverse behavior. If it was important for the traverse behavior to make it to the site by a specified time, it could have a priority that was based on the time remaining and the distance left to travel. As far as error recovery goes, if any one
of the behaviors encountered an error which it could not recover from itself, it would simply stop, and the other behaviors would take over. This basic functionality has been shown in the current system by running a roam behavior and a find interesting object behavior at the same time.

6.2.2 Preliminary Subsystem Test Experiment

As the first revision of the major subsystems of the project neared completion, a simple, yet inclusive set of simple behaviors was set up in simulation to exercise the subsystems. There were two autonomous behaviors implemented and a GUI for base control was created as well. The test was to simulate the interaction of behaviors that run independently, yet work together on a common goal- to explore an area. The first assumption made is that in general the robot should be moving forward. Hence a behavior was written that simply tries to take control of the base and then continually move it forward. The second assumption is that going in a straight line all the time is boring, so a second behavior was written to take control of the rover base at random time intervals, and to execute a turn of random direction. The third element is a GUI for a user who can monitor the motion of the base, and take control if desired.

While this experiment might sound trivial and over-simplified, it uses all the vital elements of the system, and if it could be made to work correctly, then it is only a matter of scale to implement any other system. The port system is tested as the encoders, controller, and motors are connected using ports. The arbitration system is tested in that there are three separate behaviors possibly vying for control of the base at any one time. It tests the mirror system since the GUI runs using a mirrors of the encoders, controller, and motors. Most importantly, it tests the integration of these subsystems to operation of the rover in an “orderly and effective manner,” as stated in this thesis’ introduction.
Test Execution

A script was written to instantiate BaseEncoders, SimpleBaseController, and BaseMotors, and each was added to the high-rate servoloop. A MoveForwardBehavior and RandomTurnBehavior were instantiated and added to the mid-rate servoloop. All the objects were added to the BehaviorManager. The MoveForwardBehavior was started with a priority of one, and the RandomTurnBehavior was started with a priority of two. The MoveForwardBehavior immediately took control of the SimpleBaseController and enabled it with its key. This made the simple base controller try to take control of the BaseEncoders, and BaseMotors, and upon success, it enabled them and wired the ports together as necessary. The MoveForwardBehavior then started a trajectory to move the base forward. Each time through the servoloop, the RandomTurnBehavior would generate a random number, and if it fell within a specified range, it would try to take control of the hardware. It was assumed here that the MoveForwardBehavior would execute many short trajectories rather than one long one, and therefore the RandomTurnBehavior waited until the completion of the current trajectory before submitting a new one. It could just as easily cancel the current trajectory, and execute its own. Once the turn trajectory has been successfully completed, the RandomTurnBehavior releases control of the hardware, and the MoveForwardBehavior is then able to take over again.

At any time a human user can take control of the base using a SimpleBaseControllerGUI object. Figure 6-1 shows one run of the experiment where the two behaviors successfully caused the rover to wander around aimlessly. Figure 6-2 shows what a little non-silicon based intelligence can do.

The running of these experiments brought a couple of issues to light. First is that of soft errors. These are errors that don't show up in the form of exceptions or other detectable forms. This can include things such as logic errors that make it impossible for a behavior to move from one state to the next.
Figure 6-1: One run of the Behavior Arbitration and Port usage experiment.
Figure 6-2: There was obvious human involvement here. One incarnation of the SimpleBaseControllerGUI is shown.
Walled-in Test

In another test, a new behavior was added called AvoidObstaclesBehavior. This behavior was given a priority of 100. Simulated walls were added to the edges of the workspace by monitoring the position of the simulated rover, and when one wheel left the workspace, a virtual joint limit was put on that wheel. As the current trajectory continued to move the desired encoder value further, the current limit would be reached which would cause an update to be sent out to all BaseMotors state listeners. The AvoidObstaclesBehavior which registered itself with BaseMotors as a state listener checks the current limit flag, and if it is set, it takes control of the base and immediately stops any running trajectory. It backs the rover up and then turns left or right depending on which wheel hit the wall first. The object list file for this test was:

```
rtb.motor.base.BaseEncodersGUI high 0
rtb.motor.base.BaseMotors high 0
rtb.motor.base.SimpleBaseControllerGUI high 0
rtb.motor.base.MoveForwardBehavior mid 1
rtb.motor.base.RandomTurnBehavior mid 2
rtb.motor.base.AvoidObstacleBehavior low 100
```

The object list file tells the Start script which objects to load, what servoloop rate to give them, and the priority to enable them with. A zero priority means do not enable this object on startup. The test was first run with all of the behaviors in the above list running. This produced results very similar to previous tests except that when the rover hit the wall it would back up and turn. A screenshot of this test is shown in Figure 6-3 (In previous experiments the simulated rover would wrap around in the workspace, meaning it would leave on one side, and then come back in on the other.) In another test, the RandomTurnBehavior was commented out of the object list file. This produced a very systematic search for a way out of the workspace shown
in Figure 6-4.

### 6.2.3 Test on Rover Hardware

In order to switch the behaviors over from running in simulation to running the actual rover, it was simply a matter of running them on the right computer. For instance the BaseMotors were written so that if it was instantiated on the computer with the appropriate hardware then it would use it, otherwise it would automatically start a simulation. In addition to the existing behaviors, a new one was written to visually servo the rover on colored objects.

The behaviors where run on several computers. Europa.ai.mit.edu was host to the BaseMotors, BaseEncoders, SimpleBaseController, and AvoidObstacleBehavior objects. Poincare.ai.mit.edu was host to MoveForwardBehavior, RandomTurnBehavior, TrackBallBehavior, and SimpleBaseControllerGUI. Io.ai.mit.edu was host to Camera. The different behaviors where assigned priorities as follows:
Figure 6-4: A Walled-in test with the RandomTurnBehavior disabled.

1. BaseMotors 0
2. BaseEncoders 0
3. SimpleBaseController 1
4. AvoidObstacleBehavior 100
5. MoveForwardBehavior 2
6. RandomTurnBehavior 3
7. TrackBallBehavior 10
8. SimpleBaseControllerGUI 20
9. CameraGUI 0

With all of these low-level behaviors running together, a very useful high-level behavior was produced. The rover would start out moving forward in small steps. At
random intervals, a turn of random magnitude would be executed. This produced a random walk of the playpen. If the rover hit a side or an obstacle, it would stop and back up to the left or right depending on which wheel hit first (see Figure 6-7). If a green ball came into the view of the camera at any time, the rover would move until the ball appeared in the center of the screen (see Figures 6-6 and 6-5).

Each time the test is run, the rover wanders around until it sees the green ball and then servos on it. At the time of the test, the arm hardware was not functional. However, it would be very straightforward at this point to write a new behavior that triggers when the ball is in the right place and then picks it up. The test was run several times varying the average time between random turns to see how that affected the time to find the ball. Times ranged anywhere from one to ten minutes. The rover tended to get stuck in the corners of the playpen.

One main issue this test brought to light was the fact that the latency of the network greatly affects the speed at which the overall system would run. The main reason for this is that in many cases when data was transferred over the network, no
Figure 6-6: Shown here is the rover as it visually servos on the green ball.

Figure 6-7: Shown here is the state of the BaseMotors object just before the AvoidObstacleBehavior takes over.
new threads were created. This meant that the program would have to wait for all the communication to be completed before continuing. This could be improved in a general way by modifying the Mirror class to spawn a new thread for the transferring of data. Other than that, all subsystems performed very well.
Chapter 7

Conclusions and Recommendations

7.1 Summary

This chapter presents the conclusions of this research as well as some recommendations and future work. The intent of this thesis has been to explore some new possibilities in the area of web-based control, design the supporting subsystem, and test the design on an actual robot.

7.1.1 The Arbitrated System Control Environment

The chapter on the ASCE implementation presented several subsystems, giving an overview of the theory behind each and how they work. The use of these subsystems together constitutes the “Arbitrated System Control Environment”.

One of the main products of this research is the ASCE java package which is home to the subsystems developed. Although initially the intention was to fully document every part of the system in this thesis, that was soon found to be unreasonable as the system is far to large. Instead, an attempt was made to present the main systems in sufficient detail as to be useful. These include mirror, arbitration, servoloop, port, behavior. Other members of the asce package can be explored using the on-line
7.2 Recommendations and Future Work

One area not covered in this thesis is that of the visual application builder. The design of the system with ports for data transfer lends itself very nicely to a visual layout system. A very primitive visual behavior builder was written which would allow several objects to be shown in a window. A user could use the mouse to connect inputs and outputs. This should be developed further, as it needs only slight modifications to become a very powerful tool.

Further research into the Mirror system should explore network latency issues and the most effective use of threads, as well as the possibility of using other network protocols that might lend themselves better to a particular type of data. For instance, for most tele-operation applications, a protocol that drops packets rather than resending is desirable, as the time it takes to resend often makes the data it is sending obsolete.

Another issue that needs further research is that of security for the mirror class. A user-transparent system with encryption and command source verification should be explored.

Future work will focus on setting up a public web site where people can use the system and performance data and user input can be collected. Some of the issues and features mentioned here will be explored. Specifically, an applet based system builder and additional GUI tools will be created to facilitate system monitoring.

7.3 Conclusions

This thesis has presented the design and implementation of a new framework for controlling robots via the Internet. This framework has been applied to the problem of controlling the MIT Rover Test Bed, a small wheel based mobile robot with a simple vision system and two arms. The main features of the system are that it allows for great flexibility in the actual control strategy taken. (e.g. centralized, dis-
tributed, behavior-based, etc.), and that it allows transparent access to every level of the system through any computer connected to the Internet. Due to the fact that many computers can access robot elements simultaneously, a priority based arbitration system has been included in the framework to allow the system elements to filter conflicting inputs and operate in an orderly and effective manner.
Appendix A

Glossary

1. Class Reflection - The ability to inspect a class object at run-time and find out about its members and methods.

2. Exception - a Java class that represents some error condition in a running program. An Exception is “thrown” when the error occurs, and then can be “caught” by part of the program designed to deal with such an error.

3. FEGs - Fast Eyes Gimbals, the 2dof cable-driven camera pointers used in conjunction with the WAM.

4. FTP - File Transfer Protocol, a utility used to transfer files between computers across the network.

5. GUI - Graphical User Interface, a way to provide information and options to users in a graphical format often using buttons, sliders, etc.

6. HTTP - Hyper Text Transfer Protocol, a way to transfer hyper-text documents across the web usually for viewing directly in a web browser such as Netscape Navigator.

7. Remote Object - Any java object instantiated from a class that implements a remote interface.
8. RMI - Remote Method Invocation, a specification in the Java programming language that allows methods of a object running on computer A to be invoked by computer B across the network.

9. RTB - Rover Test Bed, the highly generic name given to the rover robot developed using the system presented in this thesis.

10. WAM - Whole Arm Manipulator, a four degree-of-freedom cable-driven robot designed by Bill Townsend in 198? with a 3dof endeffector called the Talon designed and built by Akhil Madhani in 1995.
Appendix B

Creating a Mirrored Object

B.0.1 How to create a mirror class.

In order for an object to be a mirror, it must:

1. instantiate an asce.mirror.Mirror helper object.
2. implement asce.mirror.Mirrored.
3. execute the asce.mirror.Mirror:outputState method when its state has changed.
4. use the asce.mirror.Mirror:executeParentMethod method when a local user executes a child’s method that could change it’s parents state.

Once it meets these requirements it is a fully functional mirror. A parent mirror uses OutputState to inform children of a change in it’s state. A child uses ExecuteParentMethod to execute one of it’s parents methods. With these two methods, a class has all the tools it needs to create mirrored objects which provide the illusion to the user of having the root object running on the local host (with some latency of course). These requirements are each explained in some detail in the following sections.

Instantiating asce.mirror.Mirror

The main thing to be aware of when instantiating asce.mirror.Mirror is that the mirror by itself does nothing until it is connected to another mirror, (its parent), or
until another mirror connects to it. A mirror can only have one parent but may have several children. There is only one constructor for the mirror class, and the argument requires an asce.mirror.Mirrored object, this should be the object that is instantiating the mirror, for example:

```java
myMirror = new Mirror(this);
```

At this point the mirror is not connected to anything else and is not “visible” on the network. To make the mirror visible on the network, it is necessary to “bind” it in the rmiregistry. The rmiregistry is a java remote method invocation name server which will return the remote object associated with a String name, and it must be running on the machine for mirrors to operate. To start the rmiregistry, on Suns type:

```
rmiregistry 
```

and on NT machines type:

```
start rmiregistry 
```

To register or bind your mirror in the rmi registry you use the following form:

```java
myMirror.bind('''myObjectName''');
```

Of course the object name must be unique to the computer it is running on. Once the object is registered, it becomes a viable parent, meaning that other mirrors can connect to it and become its children. If for any reason an object needs to unregister itself from the registry, it can use the asce.mirror.Mirror:unbind method, but note that this will not disconnect any children mirrors that had previously connected.

If the mirror is also child (rather than a root, roots have no parents), then it must connect to its parent. It does this using the connectToParent(String pname) method. This boolean method returns false if the connection is not made. The string representing the parent’s name must take a precise form:

```java
myMirror.connectToParent('''/hostname/objectname''');
```

eg.

```java
myMirror.connectToParent('''/europe.ai.mit.edu/BaseMotors''');
```
The hostname is optional if the parent is running on the same host as the child. The disconnectFromParent method will remove this mirror from the childlist of the parent.

**The OutputState Method**

A parent uses OutputState to inform children of a change in it’s state. Acting as a parent, whenever a mirror feels its state has changed sufficiently to warrant an update of its children, it needs to call super.OutputState(state). In practice the “super” can usually be left out unless the subclass defines its own OutputState method. The argument is of type java.io.Serializable, and should be an object which contains all the data that completely describes the state of the object. For C and C++ users, this is basically a struct with all the variables that describe the state of the object. The easiest way to set this up is for every object to also have an objectState. For example, the BaseEncoders contains a BaseEncoderState object which has the following definition:

```java
//--------------------------------------
package rtb.motor.base;

public class BaseEncoderState
    implements java.io.Serializable
{
    public BaseFlags flags = new BaseFlags();
    public long time = System.currentTimeMillis();
    public int[] pos = new int[2];
    public int[] vel = new int[2];
}
//--------------------------------------
```

Notice that the state object must implement java.io.Serializable interface. This interface doesn’t have any methods, and just serves to mark object which can be seri-
alized for saving to disk or transmission over the network. This object doesn’t require any special encoding to serialize it (since all its member variables are simple types), therefore all that is needed is to include the line, “implements java.io.Serializable.” More information on serialization is available online. (See Appendix D.2)

The OutputState method notifies all of the children registered with the mirror passing them the new state variable. Each child mirror will then cast itself to type MirrorSubclass and call the mirrorStateChange method. This is the
Appendix C

Code for Selected Classes and Interfaces

This appendix list several of the interfaces and classes which are refered to in the body of the thesis. Please forgive the inclusion of so much computer code. A first attempt was made to include only small snippets that where referenced throughout the thesis, but that was found to be an editing and formatting nightmare. Instead it was decided to include the complete code for only the more vital of the classes and interfaces.

As this is code is all very much in the development stage, it is likely that several bugs will be found, and the author can take no responsibility for the results of using the code. Also note that everything contained in this thesis is copywrited, including the following code, and therefore unauthorized use or reproduction is prohibited. If there are specific questions regarding this code or the use there of, feel free to contact MIT, or the author at theo.ai.mit.edu.
C.1 asce.mirror.Mirrored

*/

package asce.mirror;

public interface Mirrored
{
    void mirrorStateChange(java.io.Serializable state);
}

C.2 asce.mirror.Mirror

package asce.mirror;

import java.lang.reflect.*;
import java.rmi.*;
import java.util.*;

/**
 * The Mirror class is the workhorse for all network communications in the asce package. A mirror is a helper object which allows two way communication between objects running in different processes or especially on different machines. For an object to mirror another it must implement the Mirrored interface, and instantiate a new mirror object. To allow children mirrors to connect to it, it must bind itself in the rmiregistry with its name. This
of course requires that an rmiregistry is running
on the machine hosting the object to be mirrored
and that it is registered properly. To connect to
a parent mirror it must supply the fully qualified
name of the object to be mirrored. Parent mirrors
are responsible to pass any significant changes in
state along to the children using outputstate. All
mirrored also must properly handle any method
executions which would change the state of the
object and pass them to the parent mirror, until
they reach the root where they can be properly
executed and then the changes to the state will
filter down through the children. For further
information on Mirrors refer to "Arbitrated Robot
Control on the Web, System Design and Implementation"
by Daniel Theobald.

/**
public class Mirror
extends java.rmi.server.UnicastRemoteObject
implements MirrorRMI
{
/** The name by which this object will be referenced in the registry. */
private String name = null;
/** The name of this mirrors parent object, "hostname:objectname" */
private String parentName = null;
/** This mirrors parent remote reference */
private MirrorRMI parent = null;
/** A Vector of remote references to this mirrors children */
private Vector children = null;
/** The most recent state of the parent object */
protected java.io.Serializable state;
/**
 * It is the root if it has no parent. The default is root, but
 * once the connectToParent method has been called, then it is no
 * longer root, and cannot be made root.
 */
private boolean root = true;
/** The object that uses this mirror */
private Mirrored mirrored;

/**
 * It is the root if it has no parent. The default is root, but
 * once the connectToParent method has been called, then it is no
 * longer root, and cannot be made root.
 */
public boolean isRoot(){return root;}

/**
 * This is the only constructor, the argument should be the object
 * using this mirror, which should implement the mirrored interface
 */
public Mirror(Mirrored mirrored)
     throws java.rmi.RemoteException
{
    this.mirrored = mirrored;
}

public boolean connectToParent(String pname){
    root = false;
    this.parentName = pname;
    try{
        System.out.println("About to lookup parent Mirror");
        parent = (MirrorRMI)(Naming.lookup(pname));
    }
}
} catch (Exception e) {
    System.out.println("Could not find parent: "+pname);
    System.out.println("will continue to try");
    return false;
}

try{
    System.out.println("About to register with parent Mirror");
    parent.connect((MirrorRMI)this);
    System.out.println("Registered successfully.");
} catch (Exception e) {
    System.err.println("Could not register with parent.");
    System.err.println(e);
    try{
        System.out.println("+((java.rmi.RemoteException)e).detail");
    } catch (Exception ex) {} 
    parent = null;
    return false;
}

return true;
}

public void disconnectFromParent() {
    if (parent != null) try{
        parent.disconnect((MirrorRMI)this);
        parent = null;
        parentName = null;
    } catch (Exception e) {
        System.out.println("Trouble disconnecting from parent mirror: "+e);
    }
}

/**
* If this mirror will have children, then this method
* should be called exactly once soon after the mirror
* is instantiated.
* /

public boolean bind(String name){
    try{
        Naming.rebind(name, (MirrorRMI)this);
        System.out.println(name+" Mirror bound in registry");
    } catch (Exception e) {
        System.out.println("Could not bind this, will run without");
        return false;
    }
    this.name = name;
    children = new Vector();
    return true;
}

public void unbind(){
    if(name != null){
        try{
            Naming.unbind(name);
        } catch (Exception e) {} 
        name = null;
        children = null;
    }
}

public void finalize()
{
    if(parent != null) try{
        parent.disconnect(this);
    }
public java.io.Serializable
executeMethod(String cast, String methodname, Object[] args)
    throws java.rmi.RemoteException,
            ClassNotFoundException,
            NoSuchMethodException,
            java.lang.reflect.InvocationTargetException,
            IllegalAccessException
{
    java.lang.Class cls = Class.forName(cast);
    java.lang.Class argtypes[];

    if(args == null) argtypes = null;
    else {
        argtypes = new java.lang.Class[ args.length ];
        for(int i = 0; i < args.length; i++) {
            argtypes[i] = args[i].getClass();
        }
    }

    java.lang.reflect.Method m = cls.getMethod(methodname, argtypes);
    return (java.io.Serializable)(m.invoke(mirrored, args));
}

public java.io.Serializable
executeMethod(String cast, String methodname, Object[] args, int[] sc)
    throws java.rmi.RemoteException,
            ClassNotFoundException,
            NoSuchMethodException,
            java.lang.reflect.InvocationTargetException,
IllegalAccessException
{
    java.lang.Class cls = Class.forName(cast);
    java.lang.Class argtypes[];

    if(args == null) argtypes = null;
    else{
        argtypes = new java.lang.Class[ar length];
        for(int i = 0; i < args.length; i++){
            argtypes[i] = args[i].getClass();
            for(int j = 0; j < sc[i]; j++)
                argtypes[i] = argtypes[i].getSuperclass();
        }
    }
    java.lang.reflect.Method m = cls.getMethod(methodname,argtypes);
    return (java.io.Serializable)(m.invoke(mirrored,args));
}

public java.io.Serializable
executeParentMethod(String cast,String methodname, Object[] args)
    throws NullParentException,
       RemoteExecuteException
{
    if(parent == null){
        if(parentName == null) throw new NullParentException();
        if(!connectToParent(parentName)) throw new NullParentException();
    }
    try{
        return parent.executeMethod(cast,methodname,args);
    }catch(java.rmi.UnmarshalException e){
        System.err.println(e+"\nMay have lost parent, will try again...");
        return executeParentMethod(cast,methodname, args);
catch(java.rmi.ConnectException e) {
    System.err.println(e + "\nLost connection to parent...");
    parent = null;
} catch(java.rmi.RemoteException e) {
    System.err.println(e + "\nRemoteException...");
} catch(java.lang.reflect.InvocationTargetException e) {
    throw new RemoteExecuteException((Exception)(e.getTargetException()));
} catch(Exception e) {
    throw new RemoteExecuteException(e);
}
return null;
}

public java.io.Serializable
executeParentMethod(String cast, String methodname,
Object[] args, int[] sc)
    throws NullParentException,
           RemoteExecuteException
{
    if (parent == null) {
        if (parentName == null) throw new NullParentException();
        if (!connectToParent(parentName)) throw new NullParentException();
    }
    try {
        return parent.executeMethod(cast, methodname, args, sc);
    } catch(java.rmi.UnmarshalException e) {
        System.err.println(e + "\nMay have lost parent, will try again...");
        return executeParentMethod(cast, methodname, args);
    } catch(java.rmi.ConnectException e) {
        System.err.println(e + "\nLost connection to parent..."sv);
        parent = null;
    }
}
public void connect(MirrorRMI mirror) throws java.rmi.RemoteException {
    System.out.println("registering mirror "+mirror);
    children.addElement(mirror);
    try {
        System.out.println("About to output state to new mirror.");
        System.out.println("State is: "+state);
        mirror.stateChange(state);
    } catch (java.rmi.RemoteException e) {
        System.out.println(e);
        e.printStackTrace();
        System.out.println("Tried to register mirror, no response");
        System.out.println("Non-fatal. Removing mirror: "+mirror);
        children.removeElement(mirror);
    }
}

public void
disconnect(MirrorRMI mirror)
    throws java.rmi.RemoteException
{
    //System.out.println("unregistering");
    children.removeElement(mirror);
}

/**
 * This method should be called by the subclass whenever
 * its state has changed in a significant way. It simply
 * updates all the children mirrors with the new state,
 * removing any mirrors from the list that throw an
 * exception (which should only be caused by the loss
 * of the connection.) When the child mirror detects
 * that its state has changed, it will call its
 * subclass’ mirrorStateChange method.
 */
public void
outputState(java.io.Serializable state)
{
    //make sure my state pointer is the same as
    //the subclass so that I can give the correct
    //state to new children mirrors.
    this.state = state;
    if(children == null) return;
    //now loop through all of my children mirrors
    //passing the new state to each.
    MirrorRMI temp = null;
    Enumeration list = children.elements();
    while(list.hasMoreElements()){
        temp = (MirrorRMI)(list.nextElement());
        try{
temp.stateChange(state);
}

}catch(java.rmi.RemoteException e){
   //Since stageChange only sets the state and the change
   //flag and then returns, an exception can only mean that
   //the connection was lost, in which case the only thing
   //we can do is remove the mirror and let it reregister.
   System.err.println(e+"\n Mirror Lost: "+temp);
   children.removeElement(temp);
}
}

public void
stateChange(java.io.Serializable state)
   throws java.rmi.RemoteException
{
   //first verify that this is a valid state, do this later....
   this.state = state;
   mirrored.mirrorStateChange(state);
   return;
}
}

/*

C.3 asce.arbitration.Arbitrated

*/

package asce.arbitration;

/**
   * Any object which needs to be shared among several
* other object with possible input conflicts should
* implement this interface.
*/

public interface Arbitrated
extends ArbitratorListener
{

/**
 * This method allows potential users of this object
 * to have access to its arbitrator, which can then
 * be used to gain input control of the object. This
 * means of course that this object must have an
 * arbitrator among its member variables.
 */
    Arbitrator getArbitrator();

}

//

/*

C.4 asce.arbitration.Arbitrator
*/

package asce.arbitration;

/**
 * This interface needs to be implemented by any
 * class that will act as an arbitrator.
 */

public interface Arbitrator
extends java.io.Serializable
{
    /**
     * This method can be called by the arbitrated object
     * any time that a method that requires arbitration
     * is invoked. Of course it requires that the key
     * is passed in as one of the methods arguments.
     * One important specification is that when a new
     * key becomes valid, calls to checkKey using the
     * old key should return without throwing
     * InvalidKeyException only until the new Key is
     * checked for the first time. This allows the last
     * KeyUser to continue to use the arbitrated object
     * until the new KeyUser actually asserts control
     * by using a method which calls checkKey.
     */
    void checkKey(Key key) throws InvalidKeyException;
    boolean canTake(Key key);
    long getMaxIdleTime();
    void addSubArb(Arbitrator a);
    void addKey(Key key);
    void removeKey(Key key);
    void addListener(ArbitratorListener al);
    void removeListener(ArbitratorListener al);
    Key highest();
}
//
/*
C.5 asce.port.InputPort
*/
package asce.port;

public interface InputPort
extends theo.util.Nameable{

/**
 * This method will connect this InputPort
 * to the OutputPort supplied. No compatibility
 * testing is performed (in interest of saving
 * run-time cycles) as it is expected that the
 * connecting process has previously checked
 * compatibility. If the ports are not
 * compatible, most likely a ClassCastException
 * will be the result of the first use of
 * the port since the object providing the
 * input port will have to cast the Object
 * arg to some type in order to use it.
 *
 * The second argument must be an object
 * implementing the PortManager interface.
 * this is required for three reasons:
 * 1. If the port wants to check if it
 * should allow the connection for some
 * reason, it needs to know who is requesting
 * the connection.
 * 2. So that it can make sure that a
 * call to disconnect can only be made
 * by the object that connected it in
 * the first place.
 * 3. So that the port can notify the
 * PortManager using connectionLost(InputPort)
 * in the case of the connection being lost
 */
}
* for whatever reason.
* While 1&2 are optional, 3 is very important.
* This will often be the only way a higher level
* will know that the connection has been lost.
*/
void connect(OutputPort op, PortManager pm)
    throws PortException;
/**
* This will disconnect the given OutputPort
* from this input port. See connect for a
* discussion on the use of the PortManager.
*/
void disconnect(OutputPort op, PortManager pm)
    throws PortException;
/**
* This method should be used mainly during
* programming time rather than runtime and
* is intended as a check to help a builder
* ensure that ports will be compatible
* during runtime. Test data can be obtained
* from the OutputPort getData() method which
* should always have data of the appropriate
* type.
*/
void testDataCompatibility(Object data)
    throws PortException;
}
C.6 asce.port.OutputPort

/*
package asce.port;

/**
 * This is a generic OutputPort interface used for connecting up
 * input and output methods in a general fashion.
 */
public interface OutputPort
extends theo.util.Nameable
{
    Object getData();

    // return true if data has changed since last getData
    // boolean newData();
}

C.7 asce.port.PortManager

/*
package asce.port;

public interface PortManager{
    public void connectionLost(InputPort ip);
}

/*
C.8  asce.port.PortOwner

/*
package asce.port;

public interface PortOwner
extends theo.util.Nameable
{

/**
 * Returns a Hashtable of InputPorts,
 * or an empty Hashtable if
 * this component has none.
 */
InputPort[] getInputPorts();

/**
 * Returns a Hashtable of OutputPorts,
 * or an empty Hashtable if
 * this component has none.
 */
OutputPort[] getOutputPorts();
}
//
/*

C.9  asce.behavior.Behavior

/*/  
package asce.behavior;

import asce.arbitration.*;
public interface Behavior{
    void setBehaviorManager(BehaviorManager bm);
    void enable(Key key) throws InvalidKeyException, Exception;
    void disable(Key key) throws InvalidKeyException, Exception;
}

//
Appendix D

Where to find more information.

D.1 Online docs for the RTB system

Online documentation for the RTB system can be found at:

http://www.ai.mit.edu/projects/rocky/docs/codeUC
http://www.ai.mit.edu/projects/rocky/docs/codeLF

The codeUC directory has docs for the UnderConstruction version of the software, while the codeLF directory has the LatestFunctional release.

D.2 Online docs for the Java language

Online documentation for the Java language is available at the following sites:

http://www.javasoft.com/products/jdk/1.1/docs/index.html
http://www.ai.mit.edu/projects/rocky/docs/jdk

Interfaces, classes, and methods are easily found by using the full package name to negotiate the html tree.