The Remote Interaction Planning and Visualization Tool

by

Robert Shibli Damus

B.S., Ocean Engineering (1999)
Massachusetts Institute of Technology

Submitted to the Department of Ocean Engineering
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Ocean Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

© Massachusetts Institute of Technology 2001
All rights reserved.

Signature of Author .................................................................

Department of Ocean Engineering
May 11, 2001

Certified by .................................................................
Nicholas M. Patrikalakis, Kawasaki Professor of Engineering,
Professor of Ocean Engineering and Mechanical Engineering
Thesis Supervisor

Accepted by .................................................................
Henrik Schmidt, Professor of Ocean Engineering
Chairman, Departmental Committee on Graduate Studies
The Remote Interaction Planning and Visualization Tool

by

Robert S. Damus

Submitted to the Department of Ocean Engineering
on May 7, 2001 in partial fulfillment of the requirements for the degree of
Master of Science in Ocean Engineering

Abstract
Unmanned observation of the ocean will continue to be an area where the nascent class of autonomous underwater vehicles (AUVs) need to communicate either locally or remotely with an operator for the purposes of updating mission parameters and offloading of data. Such a scenario poses several important questions to the underwater vehicle community that has been addressed in my research. This thesis investigates the evolving software technologies that will allow for large-scale operation of a multi-user environment for control of multiple vehicles. In particular, the software tool developed, known as the Remote Interaction Planning and Visualization (RIPV) tool, is written in Java and takes advantage of Java’s distributed computing capabilities.

The RIPV architecture provides for remote monitoring of and interaction with vehicles sampling in the water column, a relational database model for storing extracted vehicle data in a database repository, and a means of interfacing the visualization capabilities of Matlab. This will provide the user the ability to do on-the-fly mission planning, all from a web browser that loads a lightweight applet without installing software on their machine. A central server known as the RIPVServer handles clients by marshaling their requests to the appropriate service, called an RIPVService, that have registered with the system. The functionality of the services is made available to the client through the graphical user interface that loads in the web browser. The RIPVServer accesses the services via Remote Method Invocation (RMI), after calling methods that are made available in a public remote interface.

Three core services provide the RIPV system with its basic functionality: a service for interfacing a serial port, one that interacts with a database, and a service that produces Matlab visualizations. They are known as the SerialPortServer, the DatabaseServer, and the DataVisServer. Cross platform independence is achieved in the RIPV framework and the code is designed to be adapted and functionality extended through the introduction of new RIPVServices.

Thesis Supervisor: Nicholas M. Patrikalakis, PhD
Title: Kawasaki Professor of Engineering
Acknowledgements

Prior to my leaving undergraduate life, and without any better ideas, I applied to the Ocean Engineering graduate program here at MIT, and in my letter of intent wrote about how I would use a Web browser to control an underwater vehicle from afar. It seemed like the kind of thing that would get me into a graduate program – the idea is high tech, trendy, and would let me play with robots. Lo and behold, two years have passed, and those musings are now a reality.

This thesis represents a honing of my skill-set as a human, and was made possible in large part by interacting with my brilliant advisor, Professor Nicholas Patrikalakis. I am forever indebted to him for his keen insight into the problems that relate to this work as well as the contrast in which he would approach how I had thought about a solution. Without this pressure to deviate from my thought processes, my perpetual intellectual orbit would have remained at a fixed radius.

I also appreciate my conversations with Dr. Wonjoon Cho and Dr. Spyros Lalis, both of whom took time out of their life to offer useful advice.

Professor Chryssostomos Chryssostomidis was kind enough to support my efforts to apply to the MIT OE Masters program, and one of my first undergraduate advisors, Professor John Leonard, was also the first person to get me thinking about this topic during my senior year in his design class.

Without the assistance of Fred Baker, III, the Design Lab would cease to exist, and therefore I extend heartfelt thanks for his work in keeping the machines up and running.

I must thank my parents for the emphasis they put on education during my development, especially my mother and her tactics used to motivate me when it counted. They also had
another son whose humor-laden antics have kept me out of the loony bin, and thus I must give him some props, too.

Funding for this work was obtained in part from MIT Sea Grant's Industrial Fellowship, the Webb Research Corporation and MIT Ocean Engineering.

This thesis is dedicated to my Uncle Bob. His spirit shines brightly over my shoulder.
Table of Contents

ABSTRACT ................................................................................................................ 2

ACKNOWLEDGEMENTS .......................................................................................... 3

TABLE OF CONTENTS .............................................................................................. 5

LIST OF FIGURES ....................................................................................................... 9

LIST OF TABLES ....................................................................................................... 11

ABBREVIATIONS ..................................................................................................... 12

CHAPTER 1 .................................................................................................................... 14

INTRODUCTION ....................................................................................................... 14

1.1 COMMUNICATING WITH DATA ......................................................................... 14

1.2 MOTIVATIONS ...................................................................................................... 15

1.3 WEB SPACE .......................................................................................................... 19

CHAPTER 2 .................................................................................................................... 22

REMOTE INTERACTION PLANNING AND VISUALIZATION .................................. 22

2.1 PROBLEM STATEMENT ...................................................................................... 22

2.1.1 Vehicle Management .................................................................................. 25

2.1.2 User Management ...................................................................................... 27

2.1.3 Data Management ...................................................................................... 28

2.1.4 Heterogeneous Computing Environments ............................................... 30

2.2 THE RIPV APPROACH ...................................................................................... 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.1 Distributed Computing</td>
<td>31</td>
</tr>
<tr>
<td>2.2.2 Adaptability</td>
<td>33</td>
</tr>
<tr>
<td>2.2.3 Accessibility</td>
<td>33</td>
</tr>
<tr>
<td>2.2.3.1 Accessing as a User</td>
<td>34</td>
</tr>
<tr>
<td>2.2.3.2 Being the Host</td>
<td>35</td>
</tr>
<tr>
<td>2.2.3.3 Being a Service</td>
<td>35</td>
</tr>
<tr>
<td>2.2.3.4 Example of Being the Host, a Service, and a Client</td>
<td>35</td>
</tr>
<tr>
<td>CHAPTER 3</td>
<td>37</td>
</tr>
<tr>
<td>HIGH LEVEL PROGRAMMING METHODS</td>
<td>37</td>
</tr>
<tr>
<td>3.1 Object Oriented Programming</td>
<td>37</td>
</tr>
<tr>
<td>3.2 Java</td>
<td>37</td>
</tr>
<tr>
<td>3.3 Distributed Objects</td>
<td>38</td>
</tr>
<tr>
<td>3.4 Remote Method Invocation</td>
<td>40</td>
</tr>
<tr>
<td>CHAPTER 4</td>
<td>44</td>
</tr>
<tr>
<td>THE RIPV SYSTEM</td>
<td>44</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>44</td>
</tr>
<tr>
<td>4.1.1 The RIPVServer</td>
<td>47</td>
</tr>
<tr>
<td>4.1.2 The RIPVClient</td>
<td>50</td>
</tr>
<tr>
<td>4.1.3 RIPVClient’s Graphical User Interface layout</td>
<td>54</td>
</tr>
<tr>
<td>4.1.3.1 The COM Panel</td>
<td>56</td>
</tr>
<tr>
<td>4.1.3.2 The MAM Panel</td>
<td>57</td>
</tr>
<tr>
<td>4.1.3.3 The FIM Panel</td>
<td>58</td>
</tr>
<tr>
<td>4.1.4 The RIPVServices</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1.1: The AOSN concept off the coast of New Jersey at LEO15 ......................... 16

Figure 1.2: Slocum is an AUV designed for long duration deployment ..................... 17

Figure 2.1: Ways to implement DDP ............................................................................. 32

Figure 3.1: Considerations in programming language choice for RIPV ........................ 38

Figure 3.2: Distributed program using remote procedure paradigm .......................... 39

Figure 3.3: Competing remote method technologies .................................................. 41

Figure 4.1: The core components of the RIPV system ................................................ 44

Figure 4.2: RIPVServer conceptual layout ................................................................. 49

Figure 4.3: RIPV star network configuration ............................................................... 50

Figure 4.4: RIPVClient layout with CGI upload in Web browser ............................... 52

Figure 4.5: The ServiceLayoutPanel and the four JPanels that comprise it ............... 55

Figure 4.6: The COM Panel ......................................................................................... 56

Figure 4.7: The MAM Panel ......................................................................................... 57

Figure 4.8: The FIM Panel ......................................................................................... 58

Figure 4.9: Serial port interfacing using the java.comm package ............................... 61

Figure 4.10 SerialPortServer conceptual layout ......................................................... 63

Figure 4.11: DatabaseServer conceptual layout ......................................................... 65

Figure 4.12: DataVisServer conceptual layout ............................................................ 67

Figure 5.1: Socket connection between the RIPVServer and RIPVClient .................... 73

Figure 5.2: MailBox queues ...................................................................................... 77

Figure 5.3: RIPVServer architecture with pseudo code .............................................. 79

Figure 5.4: RIPVService architecture with pseudo code .......................................... 80

Figure 5.5: RIPVClient process management ............................................................. 82

Figure 5.6: Conceptual Schema for database storage of vehicle data ....................... 84
Figure 6.1: Successful ‘dir’ command returns directory listing from CF1..................87

Figure 6.2: Returned database queries ........................................................................88

Figure 6.3: Simple 2-D data plot....................................................................................89
List of Tables

Table 5.1: RIPV packages........................................................................................................... 83

Table 7.1: Advantages and Disadvantages of the RIPV System............................................. 92
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOSN</td>
<td>Autonomous Ocean Sampling Network</td>
</tr>
<tr>
<td>AUSI</td>
<td>Autonomous Undersea System Institute</td>
</tr>
<tr>
<td>AUV</td>
<td>Autonomous Underwater Vehicle</td>
</tr>
<tr>
<td>CGI</td>
<td>Common Gateway Interface</td>
</tr>
<tr>
<td>COM</td>
<td>COmmunications Management</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>DB</td>
<td>D shell connector size B</td>
</tr>
<tr>
<td>DAB</td>
<td>DAta Base</td>
</tr>
<tr>
<td>DAV</td>
<td>DAta Visualization</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DCOM</td>
<td>Distributed Component Object Model</td>
</tr>
<tr>
<td>DDP</td>
<td>Distributed Data Process</td>
</tr>
<tr>
<td>DIS</td>
<td>Distributed Information System</td>
</tr>
<tr>
<td>DISOP</td>
<td>Distributed Information System for Ocean Processes</td>
</tr>
<tr>
<td>EIA</td>
<td>Electronics Industry Association</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FIM</td>
<td>FIle Management</td>
</tr>
<tr>
<td>FK</td>
<td>Foreign Key</td>
</tr>
<tr>
<td>GIS</td>
<td>Geographic Information System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>JRE</td>
<td>Java Runtime Environment</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>MAM</td>
<td>MAilbox Management</td>
</tr>
<tr>
<td>MBARI</td>
<td>Monterrey Bay Aquarium Research Institute</td>
</tr>
<tr>
<td>MOOS</td>
<td>MBARI Ocean Observatory System</td>
</tr>
<tr>
<td>OLE</td>
<td>Object Linking and Embedding</td>
</tr>
<tr>
<td>ONC</td>
<td>Open Network Computing</td>
</tr>
<tr>
<td>OOP</td>
<td>Object Oriented Programming</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PK</td>
<td>Primary Key</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>RIPV</td>
<td>Remote Interaction Planning and Visualization</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>RS</td>
<td>Recommended Standard</td>
</tr>
<tr>
<td>RX</td>
<td>Receive</td>
</tr>
<tr>
<td>SGI</td>
<td>Silicon Graphics</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>TLO</td>
<td>Task Level Organization</td>
</tr>
<tr>
<td>TTL</td>
<td>Transistor Transistor Logic</td>
</tr>
<tr>
<td>TX</td>
<td>Transmit</td>
</tr>
<tr>
<td>URI</td>
<td>University of Rhode Island</td>
</tr>
<tr>
<td>VIP</td>
<td>Vehicle Instrument Platform</td>
</tr>
<tr>
<td>WRC</td>
<td>Webb Research Corporation</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Communicating with data

The current state of communication technologies has moved the concepts implicit in remote sensing of the ocean closer to becoming a reality. Before we can expect vehicles with finite storage capacities and limited behavior sets to exhibit truly autonomous adaptive analysis of a desired underwater region, exploiting the revolution of data delivery channels must be examined. This thesis investigates the evolving software technologies that will allow for large-scale operation of a multi-user environment for control of multiple vehicle instrument platforms. For such a platform to be integrated into a multi-level, multi-disciplinary application, a means of direct connection between the user and the actual ocean parameter being observed is essential. Taking this one step further, maximum value is achieved by a system that is capable of independent use by researchers located remotely from the actual site of the network conducting the ocean research. The Remote Interaction Planning and Visualization (RIPV) tool has been developed in the Java programming language to assist vehicle operators in their approach to mission planning by making it possible to manipulate data from a vehicle, store the information in a database repository, or produce visualizations products from the data, all from an interface supplied to their World Wide Web Browser.
1.2 Motivations

Operator assistance for underwater robotic missions has entered into an age where it is often not necessary to be within the locale of the vehicle to have eventual access to the parameters the vehicle is observing. NASA developed the first telerobotics platforms [NASA], and the concept gained momentum in the underwater vehicle community with the introduction of Autonomous Underwater Vehicles (AUVs) as exemplified in the Autonomous Ocean Sampling Network [Curtin 93], where a network of vehicles continually samples an area, offloading the data they gather and then receiving new instructions for action (see figure 1.1). Three communications links are seen in the final AOSN system: 1) AUV to node (via acoustic modem), 2) node to shore (via radio or satellite), and 3) shore to scientist (via the Internet). This thesis is motivated by the second and third aspects of the AOSN communications structure, as they are currently being examined and proof of the concept has been shown in experiments involving AUVs in the Haro Strait [Schmitt 96], and Buzzards Bay [Kim 96].

AOSN is an example of a network of multiple Vehicle Instrument Platforms (VIPs) where the main idea is that that a network of research vehicles and distributed acoustic and point sensors would do a better job of canvassing the ocean in order to improve upon, among other things, the spatial resolution of sampled data sets. That a research vehicle could be used as a multi-user platform is implicit in the design of the AOSN. To date, however, there is little experience associated with this endeavor because of the lack of a communication interface that is easily used by multiple users to support a shared environment for multiple distributed VIPs, as well as several gaps in the facets of the system, namely long-duration vehicles to test the validity of such interfaces [Chappel 00].
It is inherent in the design of a true AOSN that the vehicles doing the observation will do so in an unsupervised fashion, but periodically unload information at specified nodes which is then disseminated to the appropriate computational engines for evaluation and updating of mission objectives; the goal being that an operator has little or nothing to do with how the data is processed, rather they are useful in diagnosing vehicle failures or networking issues. With respect to this paradigm of operation in the ocean environment, we are far from implementing such a system [Robinson 99, Malanotte-Rizzoli 99]. Continual observation of a vehicle and its functional state is the norm, and often the only way to guarantee the validity of measurements and evaluate how the vehicle behaved in a mission. For the majority of operations that transpire where environmental data is recorded, the simple truth is that unanticipated vehicle failure (hardware, software)
occurs in some percentage of these missions, thus necessitating active intervention by vehicle operators. It follows that any tool developed to interface a broad array of vehicles must be configurable to serve such a purpose.

A likely scenario to increase the utility of a single vehicle would be the incorporation of independent users, each promoting their own research, through specified access to the onboard resources. To deal with the issue of longevity, the Webb Research Corporation has developed the first oceanic Glider, *Slocum* [Simonetti 98]. *Slocum* (figure 1.2) uses an environmentally driven engine to overcome caveats introduced by limited battery life and is a representation of the VIP paradigm for continued access by many users over a long-duration deployment. The utility of AOSN lies in having the system automatically reconfigure and organize itself based on the participants [Turner 96], but this is a feature that has yet to be successfully employed at outposts like LEO15 [Forrester 93] or in AUV related experiments [Schmidt 96, Beaufort 1,2]

![Slocum Glider](image)

*Figure 1.2: Slocum is an AUV designed for long duration deployment*

Furthermore, addressing the organizational potential of an operator has yet to be determined or evaluated, as scarce tools exists that bundle access to the vehicle and its data taken from previous missions in a common execution environment. A user is
expected to have their own copy of the appropriate information about the vehicle’s performance history (like sampling rate or past mission objectives), and this is a bottleneck for introducing more than one user to operate and successfully conduct research with or in a network of vehicles. It is possible for researchers to share this information outside of the specter of actively controlling the vehicle by placing the information in a common location that is available to all interested parties. The problem then becomes how the data should be stored and whether there is an appropriate means of searching the data. Here, the ocean community would benefit greatly from a well-defined schema to represent information taken from a vehicle as well as a means of doing data visualization.

It is very desirable to be able to actively control a vehicle or its subsystems by taking consistent and coordinated measurements to provide for the temporal and spatial resolution of data sets necessary for worthwhile interpretation of ocean phenomena [Schmitt 96]. While the artificial intelligence used by vehicles to perform missions has improved, it is by no means a match to an operator issuing commands to fulfill a mission objective [Kim 97]. An experienced operator is currently a requirement in the loop of making an AUV behave in an intelligent manner, as difficulties inherent in communicating underwater have precluded the implementation of truly autonomous networks that perform adaptive sampling [Catipovic 90]. Experiments that have posed questions about the parameters involved in doing Remote Monitoring [Kim 97, Kim 96] have used application and hardware-specific software. Work on generalized implementations of Remotely Interfacing a vehicle is being conducted [AUSI, MOOS].
Although the design and construction of AUVs varies in accordance with different criteria, some sort of communication requirements is invariably set, and this is where my research builds on new models for interfacing the vehicle. Advancements in the areas of Radio Telemetry and Satellite communications have influenced how the engineer creates a communication link with the vehicle. In addition to “jacking in” with a tether, Radio Frequency modems are being utilized (MIT’s Odyssey, MBARI’s Dorado, WRC’s Slocum). These modems have line of sight ranges up to 20nm [Freewave] thus allowing for realistic coastal zone observation swaths from a fixed location on land. Currently, there are no software tools to address this problem of on-the-fly Remote Interaction with a vehicle sampling in the water column, as differences in design and construction of a vehicle often lead to proprietary communications solutions.

The question becomes how to set up a system to monitor a vehicle, potentially by many users. A well-defined, maintainable operating structure is necessary to support such a system of users. A distributed computing framework, where there might be several machines on a network that are hosting vehicles and need to transmit their information to one or multiple users, is at the core of the answer to the question. The system protocol should be based on adaptable software tools.

1.3 Web Space

It is important to recognize the advent of the World Wide Web as an extremely useful means for allowing information to be shared in a graphical and intuitive manner among users. At its core, the Web is a static hypertext graph representation of data. Servers present this information to clients requesting the content of a page in a Web browser; the
Web has developed into the dominating platform for information systems on the Internet [Ciancarini 97]. Academia and business has embraced the Web for disseminating information. MIT’s Open Course Ware\(^1\) initiative is designed to make all of the Institute’s courses available online within ten years. Numerous labs and electronic journals now exist. Society has entered into the third wave: the age of instant access to information, bolstered by this powerful medium for content delivery.

The push towards a more dynamic space, with an increasing availability of networked, Web-accessible workstations, has begun. It has become more desirable to use the Web for dynamic, distributed applications. As an example of means of making the content dynamic include running scripts, or presenting the client an Applet. Common Gateway Interface (CGI) or Perl scripts are examples of executable programs that run on the server and can interpret certain user actions and respond appropriately. An Applet is an object written in the Java programming language that acts like a program running in the client’s browser, under control of the browser’s Java Virtual Machine.

The Web is a natural choice for means of presenting remotely-located users with an interface to distributed applications. Coupling the interface design with server side resources has increased the utility of dynamic content presentation. Projects that provide the user with an interface to Manage distributed sub-systems like Poseidon [Poseidon], Thetis [Thetis], or ARION [Arion] are models for how researchers can interact with information, perform useful calculations and then produce data products for interrogation. The goal of these tools is to capitalize on the diversity of resources that may be presented

to the user via distributed network coverage thereby giving the individual investigator a new means of examining their problem.
Chapter 2

Remote Interaction Planning and Visualization

2.1 Problem Statement

It has been several years since the concepts for the AOSN network were proposed, and incremental developments (Haro Strait\(^2\), Labrador Sea\(^3\), Buzzards Bay\(^4\), Arctic Ice\(^5\)) have fostered the growth of the idea. Advances in the computational resources, underwater acoustic telemetry, and methodologies for organization and localization of multiple AUVs have laid the foundation for the framework that will someday be in place for a real-time oceanographic sampling network that assists in climatologic forecasting and is a test-bed for future incarnations of ocean monitoring.

The crux of the system relies on robots, which are limited by their sensor suite; a robot’s ability to perceive their environment and act intelligently is a function of its sensors. The key observation here is that by in large robots are still only as capable as the operator who has planned their mission, if that. Certainly forages into behavior based hierarchical layered control [Bellingham 91, Brooks 99] have furthered the state of the art, but are not in the clear when it comes to having an autonomous robot make well-informed decisions.

\(^2\) Schmidt 96
\(^3\) Visbeck 94
\(^4\) Kim 96
\(^5\) Bellingham 95
about what it should do next. This particular job is accounted for in the AOSN architecture after the vehicle has uploaded its data sets to shore via some telemetry link. The shore node (i.e. a human operator) relays new instruction sets to the vehicle based on having interpreted the data and decided on a new plan of action. This begs the question of how informed the shore node is, in particular the scientist monitoring the vehicle, for making the vehicle’s next set of decisions that comprise a mission. The goal is, of course, to produce robots that make intelligent conjectures about their behavior plan based on the data it is sampling thereby removing the human from the control loop entirely. Without a robust means of perception, the robot is at the mercy of the coarse resolution of its sensors for decision-making within its environment. This issue motivates the need for visualization of data sets to improve upon mission objectives.

And finally, by noting that these vehicles have limited memory, at best limited intelligent decisions without assistance about what to do when a mission has terminated can be expected for the time being.

If well-formulated hypotheses about the data that has been collected are encapsulated and executed as missions, the utility of a vehicle in the network has increased, for instance, through conservation of energy; vehicle resources are not wasted on observing insignificant phenomena. The problem really lies in how such well-formulated decisions get proposed and then executed. Do multiple copies of the data sets exist and if so, how easy is it to disseminate this information to other scientists for their input? All researchers involved would benefit from a definitive way to store the extracted environmental data so queries on the data sets return useful results. An iterative approach to optimizing a vehicle’s mission plan and better understanding of the ocean parameters it is observing is the goal. As for the execution aspect, what provisions are made for
researchers that have an interest in the mission parameters of a vehicle, but are not physically located at the shore node?

Thus, another problem addressed by this thesis has to do with interfacing a collection of resources. For the purpose of this work, the term data device (or simply device), and resource will be used interchangeably to mean an instrument that has useful information or data that we wish to access. Examples of such devices are: robotic underwater vehicles, computer workstations or databases. Therefore, the questions outlined above include representation of resources, application specific protocols, and underlying control architecture to support a means of facilitating data extraction. From a utility standpoint, the important issues include: a means of presenting the device’s information to the user, giving them the ability to interact with and then evaluate the data they have retrieved.

To better define the approach taken to render the RIPV solution to the aforementioned problems, four specific areas were examined. First, we will look at Vehicles and their design to gain insight into what common features will make the RIPV applicable. Second, the general problem of multiple Users accessing resources concurrently will be addressed. Third, Data Management is discussed, as this is of interest for researchers extracting information from a vehicle. And fourth, the problems associated with a heterogeneous computing environment are examined.
2.1.1 Vehicle Management

Two key aspects of managing multiple robots operating in an environment that you do not have specific control over are: *who* is available to communicate with and *how* do I pass information to that object. It is often desirable to make these decisions *a priori* in order to guarantee that VIPs in the network will be able to receive commands, and are uniquely identifiable. Who is available on a network is more a task for the operator in situations where they exhibit some form of control over the behavior of the network, however consideration should be given to scenarios where a previously unidentified node registers on the vehicle network to receive commands [Turner 96].

The type of telemetry hardware configuration that will be utilized is an essential part in determining how access to a platform is established. This obviously leads to limitations in how you can deploy your network, as changes made at the hardware level must be reflected in the configuration of all vehicles that you wish to have access to. A way to expedite the solution to the problem is at the conceptual level make the software capable of handling a multitude of connection channels, like the ability to interface a vehicle with TCP/IP (via Sockets and an Ethernet cable) as well as serially (with a serial communications cable). Thus, hardware overhead becomes less of a concern for maintaining semblance in future implementations of a vehicle.

Robots inherently use some sort of processor for running the control software, and at a high level, the communications command structure that an operator has access to is a function of the underlying operating system. Commands that work in a PicoDos shell may not be valid in a PC/104 running the QNX4 environment. But at a low level, the
problem is reduced to sending bytes, or voltages, across an I/O pin that are interpreted by
the microcontroller itself and an action ensues if the appropriate command is given. For
instance, to get a listing of the files in a directory in PicoDos, a user inputs the characters
‘dir’. Similarly, ‘ls’ accomplishes the same goal in QNX.

Whatever the means used to read or write bytes to the I/O pins, there are certain
procedures that need to be followed, and a protocol is defined in terms of what types of
logic levels (at the physical layer) will be used (i.e. RS-232, RS-485, TTL) and how the
bits that comprise the bytes will be put on the pin (i.e. in a Serial fashion, one bit
following the next, or in a Parallel fashion, where several bits are placed on adjacent pins
and then read simultaneously). RS-232 is used extensively for communication between
computers and terminals at standardized data rates ranging from 110 to 921600 bits per
second [Horowitz 98]. The most common port used to connect the computer and
terminal for RS-232 communications is called a serial port, in reference to how the bytes
appear on the Transmit (TX) and Receive (RX) pins.

Choosing how to interface the microprocessor onboard a vehicle stems from the type of
flexibility on-shore or in the water that you desire. Whether you build a tether or use a
modem, the options for sending and receiving bytes to the majority of microprocessors
capable of controlling an underwater vehicle is in a serial fashion with the RS-232
protocol. Used in virtually all microprocessor systems and larger systems as well lends
credence to the prevalence of RS-232, even some 30 years after being published as a
standard by the EIA [Nichols 82].
2.1.2 User Management

By removing the user from operating a vehicle within close proximity and introducing vehicles into a multiple user environment, you must be prepared to deal with concurrent access to the resources on a user-by-user basis. Severe complications will result if commands are intercepted before their completion or if there is no way of determining if a command was received. File transfer protocols like XMODEM or packet transfer protocols like TCP/IP have a well-defined set of procedures that take place on the Sender and Receiver in order to guarantee continuity [Hughes 99, Comer 96].

The issue of dealing with the users and determining how to handle their requests to fulfill mission requirements is considered part of Task Level Organization (TLO). A myriad of possibilities exist for implementing TLO; a common solution is based on simple hierarchical TLOs [Turner 96].

Scheduling the user’s requests is also an area where details can become increasingly complex. Task scheduling is usually characterized in many applications as being $NP$ hard, however the complexity arises in the model for determining when to execute tasks. For situations where temporal constraints are relaxed, and only a logical precedence is a factor, there exists solutions to the problem that are not computationally intensive and retain the intention of the user [Jones 98].

Keeping track of the connections held by a user is best delegated to individual processes. Users exhibit different behavior patterns when accessing resources, and thus it is desirable to act based on their actions as opposed to anticipating or setting guidelines on
how you expect their behavior to be. When a user’s task is being executed, it is important that other parts of the system do not overstep their bounds and interfere with the current process. Inter-process communication is therefore needed.

2.1.3 Data Management

Information stored in a network is valuable when it can be ascertained and evaluated. One of the arguments for using AUV’s to collect data for research is that they are cost effective and make obsolete large reconnaissance platforms. But as you trade down in size for economics [Bellingham 94], you present yourself with less room to store the information that sensors are gathering. Hard drive disks are a source of heat (which can degrade electrical performance, create noise, etc), and as a result, flash cards are an option for autonomous vehicle designers with capacities that currently only provide in the low hundreds of Mega Bytes MB [Persistor, Onset]. Contrast this with the upper end of high-resolution sensors, like sonar imaging hardware, which can often produce MB of data in a matter of minutes and you are invariably strapped for storage space. Compensation can take place in the form of reduced sampling rates, altering of mission plans, periodic offloading of data at specified nodes or surfacing and telemetering via modem or satellite. The first two options listed are often unwanted solutions; the last two are feasible, the latter of which is more flexible.

Data is generally stored in a File format for ease of packaging, transfer, and execution of code, but streaming data or packets are also acceptable means of sending information. For the purposes of AUV research, executable binary code and vehicle logs are stored as files. This way, vehicle information can be ascertained by parsing the log, and new
control software can be downloaded with file transfer protocols that are inherent in the vehicle O/S. For instance, when the controlling software for the Odyssey IIb and Slocum is initiated for a new mission, the code first reads from an updateable ASCII configuration file to learn the value of new mission parameters, if any.

Important to the user is the ability to extract the data files generated, interpret logs, update the software on the vehicle, send commands back to the vehicle about what type of behavior is desired. This requires that a means of transporting a file to/from the vehicle is present. For a user to be effective, they must be able to interpret the results in some fashion. Their options then include either some sort of parsing the data files for inspection, formatting the data for output to a model or prediction system (like HOPS\textsuperscript{6}), performing a visualization using a software tool, or storing the file itself for perusal at a later time. Retrieving the information stored can be challenging, especially without well-defined directory structures or Metadata about the contents of the files that you are trying to recover. For an implementation of a web accessible ontology Metadata creator for the Poseidon DISOP, see [Wariyapola 99].

A Database Management System (DBMS) provides an interface to the tools for storing and algorithms necessary for comprehensive searches on data. A simple and logical view of a database system is the relational data model in which the user perceives the data to be organized in tabular form [Elmasri 94]. As a result, relational databases (RDBMS) are being used overwhelmingly in industry today [Sunderraman 99]. The relational data model has a strong foundation based on set theory where each attribute name is

\textsuperscript{6} Harvard Ocean Prediction System
associated with a *domain*, or set of values. The Structured Query Language (SQL) is a universal language that can be used to define, query, update, and manage a relational database [SQL].

### 2.1.4 Heterogeneous Computing Environments

The extreme heterogeneity of dispersed computational resources poses an additional problem in terms of the compatibility and translation between processors, operating systems, implementation languages, data formats, etc [Patrikalakis 99]. Establishing a static architecture for interaction fails to provide ease of access for future components that might be introduced to add functionality to the system. There is greater overhead for the developer of the system when having to interface inconsistent parts, too.

A functional requirement that is becoming increasingly prevalent in the design of web-based systems (for instance, Visualization Systems) is not downloading system specific software to a user’s machine for execution. In essence, the *client* part of the system has to be independent of the client’s computer platform [Bender 00]. Technologies that can overcome deficits imposed by different operating systems interacting in a network include the programming language Java or the CORBA standard, both of which are described in more detail in chapter 3.

### 2.2 The RIPV Approach

The approach taken to handle the problems outlined in section 2.1 can be summarized as follows: create an adaptable, teleoperation software system via the Web by incorporating a distributed computing framework to facilitate extraction of vehicle data for
communication, visualization and storage. This is known as the Remote Interaction Planning and Visualization (RIPV) system and is the core software contribution of this thesis research. In the context of the ocean sciences, the RIPV can be categorized as a Distributed Information System for Ocean Processes (DISOP), as it satisfies four criterion exactly by: making transmission transparent, uses distributed computational resources, improves data reduction and interpretation, and increases the potential for autonomous operations [Patrikalakis 99]. For a description of other projects that use distributed applications, see [ARION, Poseidon, Thetis, Robinson 00, Bender 00].

2.2.1 Distributed Computing

Distributed computing solutions are commonplace today and are used to accommodate a wide variety of user needs and applications. Distributed Data Processing (DDP) is defined according to [Black 87]:

- In contrast to the conventional single, centralized site, processing is organized around multiple processing elements (PEs). A PE is a computer or other device capable of performing automated, intelligent functions.
- PEs are organized on a functional or geographic basis.
- Distributed elements cooperate in the support of user requirements.
- Connection of processing elements is through common carriers or private links.
- It entails the dispersion of hardware and/or software and/or data.

Arguments for implementing such a system include:

- Generalization of a machine concept often leads to overkill in designing a solution.
• Specialized machines can perform fewer functions, but perform them very well.
• Allow users and developers to gain some control of the computing power in the organization of a system.

Disadvantages of such a system should not be overlooked:
• Loss of control leads to emergence of multiple standards within the organization
• Duplication of software resources might lead to conflicts in implementation
• Incompatibilities associated with introducing different versions of the software

There are several ways that DDP can be implemented:

As seen in Figure 2.1, the three basic types of a data distribution are Centralized, Partitioned, and Replicated. The RIPV tool supports Centralized and Replicated versions of the DDP. A Centralized approach seemingly defeats the purpose of introducing a distributed framework, but allows for ease of use and provides the user with the ability to launch the software package from their own server. The Replicated configuration is how the RIPV system best utilizes remote resources and takes full advantage of remote method invocation (RMI), to be discussed in section 3.4.
The RIPV tool set makes use of independent Service modules that perform as distributed PEs over a network that is accessible via the TCP/IP protocol. In addition to the Disadvantages listed above, the problems associated with accessing the same resource can lead to a situation where a user is accessing content while another user has a handle to the same reference. To prevent this failure mode, the RIPV tool supports a simple procedure known as lockout. Lockout prevents two users sharing the same resource from interfering with one another by means of inter-process communications designed so that only one user at a time can access variables. An overview of the system is given in section 4.1.

2.2.2 Adaptability

Crucial to the observation that changing computing environments often wreak havoc on software implementations that are forward compatible, the RIPV is a framework that promotes “code reuse and incremental development through object-oriented extensions for the programmer” [Lalis 00]. By virtue of the fact that the code is written in an Object Oriented Programming language, Java, a developer wishing to improve the system will be capable of doing so after understanding the layout and class relationships. This is explained further in section 3.1. Also, the code has a searchable Application Programming Interface (API) located at http://deslab.mit.edu/~rdamus/ripv_api to allow for easy reference of relevant classes for either adding to the system or designing new parts to be included in the current version.

2.2.3 Accessibility

There are three fundamental ways to access and/or implement the RIPV software:
• Solely as a *user* (or *client*), known as the RIPVClient

• Solely as the *host* of the RIPVServer

• Solely as the *provider* of an RIPVService

Each one of these ways can be coupled with the other to produce a total of 7 combinations for how the software can be run. For instance, you might want to host the RIPVServer, but also access the system as a client, or you may offer a service to the system and want to access the system to take advantage of other Services running remotely on the network. Or you may be the host and also act as a provider. The 7 possibilities stem from having 3 options for atomic pieces of the system and you could have any combination of them and any number of them on the same machine:

\[ \sum_{i=1}^{3} \binom{3}{i} = 7 \]

### 2.2.3.1 Accessing as a User

Accessing the functionality provided by the *Replicated*, or most likely method of accessing the *RIPV* system, the user must simply have:

- An internet connection
- Web Browser

Keeping with the model that no software should run on the client’s machine, the Graphical User Interface (GUI) for the *RIPV* tool is presented in a user’s Web browser. To run the software, you must be using a Java-compatible browser that runs the Java Plug-In ver 1.2 or higher (latest release is plug in 1.3)\(^7\) include:

- Netscape Navigator\textsuperscript{8} version 4.08 and higher
- Internet Explorer version 5\textsuperscript{9} and higher
- Hot Java enabled Browsers

\subsection{2.2.3.2 Being the Host}

The software is launched as the server from a machine that supports:

- A Web Server
- Internet Connection
- Java Virtual Machine

\subsection{2.2.3.3 Being a Service}

In order to provide a service to the system, you must have:

- An Internet connection
- Java Virtual Machine

\subsection{2.2.3.4 Example of Being the Host, a Service, and a Client}

If the user wishes to exercise the \textit{Centralized DDP} option by running the whole \textit{RIPV} system on a single machine, thereby essentially acting like the host machine, and a provider, but also being able to retain the functionality that a client sees then the

prerequisites are:

---

\textsuperscript{1} http://home.netscape.com

\textsuperscript{8} http://www.microsoft.com/windows/ie/default.htm

35
• An Internet Connection
  o JVM

• A Web Server
  o JVM

• A Web Browser
  o JVM
Chapter 3

High Level Programming Methods

3.1 Object Oriented Programming

As a practical and adaptable means of solving the problems addressed in this thesis, the software paradigm chosen to express the constructs and organizations is of an object-oriented flavor. This means that specific objects, or classes as they are called in Java, can be defined by code and then given attributes and functionality in the form of variables and methods, respectively. The convoluted nature of this problem lends itself nicely to this abstraction methodology, where classes can inherit, or extend characteristics from other classes in order to group the problem into manageable chunks that can be dealt with in a predictable and easy to improve upon manner.

3.2 Java

Java is a trademark of Sun Microsystems, who describe Java as a “simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded and dynamic language.”¹⁰ Java is based on the C++ programming language, but can also be used to create executable content that can be distributed through networks. It is a relatively young language [Mihanetzis 99] and this version of software was written in what is now being called the Java 2 platform, using the Standard Development Kit¹¹. Java is different from other software systems in that it’s a software-only platform that runs on top of other, hardware based platforms [Campione

¹⁰ http://java.sun.com
¹¹ http://java.sun.com/j2sc/
It is both a programming language and a platform. Java code is first compiled and turned into bytecodes, or platform independent instruction codes, and then run on an interpreter. The two major components of the Java language are:

- The Java Virtual Machine (JVM) interprets bytecodes. Each java bytecode instruction is interpreted every time the program is run on a computer.
- The Java Application Programming Interface (API) which is a collection of ready-made software components that are grouped in libraries, or packages.

When choosing a programming language to design a software system, it is important to consider the scope of the problem and whether performance issues will be of concern.

With Java the type of operating loads expected for computationally intensive applications might degrade performance, however for the RIPV problem statement, Java will perform just fine [Boszormenyi 99]. See Figure 3.1 for more detail about the decisions affecting the programming language. Fortran was not considered for this application.

<table>
<thead>
<tr>
<th>Product</th>
<th>Platform</th>
<th>Type</th>
<th>Remote Methods</th>
<th>Web Interface</th>
<th>Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA</td>
<td>Independent</td>
<td>OOP</td>
<td>RMI CORBA JNI</td>
<td>Applet, dynamic</td>
<td>Easy</td>
</tr>
<tr>
<td>C++</td>
<td>Dependent</td>
<td>OOP</td>
<td>RPC CORBA</td>
<td>CGI, static</td>
<td>Difficult</td>
</tr>
<tr>
<td>C</td>
<td>Dependent</td>
<td>Serial</td>
<td>N/A</td>
<td>CGI, static</td>
<td>Difficult</td>
</tr>
</tbody>
</table>

*Figure 3.1: Considerations in programming language choice for RIPV*

### 3.3 Distributed Objects

By distributed objects, we are referring to objects that reside in separate address spaces and whose methods can be subject to remote method calls (a *remote method call* is issued in an address space separate from the address space where the target object resides)
[Plasil 98]. Convention has deemed the code issuing the call as the client (cf with the client from the client-server architecture described in chapter 4); the target object is referred to as the server (hence the suffix naming convention for RIPVServices, see Appendix A) or simply the remote object. One of the keys of this framework is making it transparent to the user how “far” away a remote resource actually is: perhaps they reside on the same machine, are on different nodes of a network, or even different URLs. Implicit in delivering a remote method call over a network communication infrastructure is the request (asking the service) and response (bringing results back to the client).

To help programmers design and understand client-server interaction, researchers have devised a conceptual framework for building distributed programs known as the remote procedure call model or RPC model [Comer 96]. This model hinges on an application-oriented design where a solution to the problem is crafted as a single program entity and then divided at the procedural level and distributed among a group of computers to run the code. This ensures that the program can solve the problem, and does not bog the designer down with concerns about the communication aspects of the code. A conventional program in this fashion consists of one or more procedures, usually arranged in a hierarchy of calls. Below is an example taken from [Comer 96].

![Diagram](image.png)

Figure 3.2: Distributed program using remote procedure paradigm
3.4 Remote Method Invocation

Remote Method Invocation (RMI) is the pure Java way of dealing with distributed objects. Java also makes provisions for employing the CORBA IIOP protocol [Hughes 99, Plasil 98]. RMI is a high-level distributed-object interface to networking. This means that instead of having to worry about low-level calls to opening sockets and establishing network connections yourself, an underlying networking infrastructure handles the networking for you. RMI is used to invoke remote methods on remote objects as though the methods were residing locally on the machine being used. It can be used in client-server type architectures, as well as in a peer-to-peer relationship, thus blurring the line between server and client. This is useful, for instance, in creating a low overhead process like an event server where client notification is not continuous. Currently (and for the near future), RMI can only work on Java enabled machines, thus limiting its usefulness in implementation for legacy code. The aid of a Java Native Interface wrapper, which is akin to adding a layer of code on top of existing software, provides an interface to procedures that already have been written, thus extending the breadth of RMI’s domain in legacy systems.

RMI is a competing technology for developing enterprise related business solutions in a distributed computing environment and is desirable when invoking a method from a remote machine makes more sense than having the method reside locally. RMI can be used to pass objects across a network, the whole purpose of RMI being: allow Java objects to communicate via their methods regardless of where the applications reside. RMI allows the passing of behavioral objects, which implies passing both the object’s state and behavior. An object’s behavior is provided by its class file, which is
automatically downloaded by the recipient. RMI provides for safety considerations in the form of an RMISecurityManager.

Competing technologies with RMI are depicted in Figure 3.3. CORBA, the Common Object Request Broker Architecture, is a highly versatile and powerful architecture for developing distributed object environments\(^{12}\). It has the benefits of being widely accepted and is gaining momentum, but its complexities due to the general nature of its organization precluded its use in this project. DCOM, Microsoft’s protocol that enables software components to communicate directly over a network stands for the Distributed Component Object Model. It is not platform independent and thus was not considered.

<table>
<thead>
<tr>
<th>Product</th>
<th>Platform</th>
<th>Interface Language</th>
<th>Transaction Speed</th>
<th>Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>Independent</td>
<td>java.rmi.Remote</td>
<td>Fast</td>
<td>Easy</td>
</tr>
<tr>
<td>CORBA</td>
<td>Dependent</td>
<td>IDL</td>
<td>Faster</td>
<td>Depends</td>
</tr>
<tr>
<td>DCOM</td>
<td>Dependent</td>
<td>OLE</td>
<td>Faster</td>
<td>Difficult</td>
</tr>
</tbody>
</table>

*Figure 3.3: Competing remote method technologies\(^{13}\)*

For a client and server to communicate via remote method calls, the party that will be accessed remotely must make a collection of these methods, known as an interface (that extends the RMI super interface, java.rmi.Remote\(^{14}\), publicly available. But in order for the methods in the interface to be called, one must have a local reference to the remote object whose interface you are accessing the methods of. Typically, a client will be able to establish a remote reference by querying (in Java this is known as doing a lookup\(^{15}\))

\(^{12}\) The Object Management Group formed to promote the use of CORBA. For more info visit [http://www.omg.org](http://www.omg.org)

\(^{13}\) Plasil 98


on the machine where the remote object resides. The lookup takes place in the registry, called the \textit{rmiregistry}, which is a table that keeps track of what remote object have been \textit{bound} (via a call to the \texttt{Naming.bind()} method when the remote object first is instantiated by starting its application). A reference to the remote object is returned to the client as a result of the lookup, and this then allows the client to gainfully employ the methods defined in the remote object's interface as though this remote object was actually residing in local memory space.

For the \textit{RIPV} system, the interface associated with RMI calls between the RIPVServer and RIPVServices is the \textit{GenericRMIInterface}, which defines three principle methods, \texttt{serviceRequest()}, \texttt{serviceReply()}, and \texttt{register()}\footnote{http://java.sun.com/j2se/1.3/docs/api/java/rmi/Naming.html#bind(java.lang.String, java.rmi.Remote)} (see footnote for the other methods defined). Implementing a server object's interface is a way of designating that object as providing the \textit{service} associated with the methods in the interface. Thus, the \textit{GenericRMIInterface} is the set of methods that all services implement and are capable of providing as a service to the RIPVServer (in this case the RIPVServer is acting as the client to the RIPVService). Important to note is the RIPVServer exposes the RIPVServices to the RIPVClients, thus making the RIPVServices function as servers with respect to the RIPVClients. Another way to think about this is to consider the RIPVServer as the \textit{broker} for the system, keeping tabs on services, and then presenting them to users who want to gain access to what the services offer. However, since the RIPVServer also implements the \textit{GenericRMIInterface} (actually, a separate reference to

\begin{footnotesize}
\begin{itemize}
\item[\footnotesize17] http://deslab.mit.edu/~rdamus/ripv_api
\end{itemize}
\end{footnotesize}
the service, the ServiceReferences that are stored in a Vector\textsuperscript{18} residing at the
RIPVServer implement the interface), the RIPVServer is capable of being a service with
respect to the RIPVService, thereby utilizing a feature of remote methods known as
callback [Hughes 99]. Callback is a useful tool for treating both the client and server as
both types of remote objects simultaneously.

Launching a system that uses RMI requires understanding how the request and reply are
handled by the underlying framework. These actions are handled by specific pieces of
code that are generated after compiling, called the \textit{Stub} and the \textit{Skeleton}\textsuperscript{19}. The stub is
distributed among the clients so that their RMI code can successfully call on the server.
And the server maintains the skeleton, which is responsible for handling the incoming
RMI calls from the client. For actual deployment, the client in this case, the RIPVServer,
wishes to access the services provided by the remote RIPVService objects, and does so
by having a collection of the RIPVService stubs so that it can gain a remote reference via
the lookup in the appropriate rmiregistry. Similarly, when the RIPVService acts as the
client to the RIPVServer by registering itself using the \textit{register()} method, it must have a
RIPVServer stub to guarantee that its RMI calls are marshaled correctly through the
network.

\textsuperscript{18} http://java.sun.com/j2se/1.3/docs/api/java/util/Vector.html
\textsuperscript{19} http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html#RMISTubsAndSkeletons
Chapter 4

The RIPV System

4.1 Overview

The *Remote Interaction Planning and Visualization* tool is a *client-server* based software system designed in the Java programming language that serves up *nonstandard*\(^{20}\) *application services* [Comer 96] for the purposes of gaining access to a resource located *remotely* with respect to the network hosting the server\(^{21}\). As depicted in figure 4.1, the core components of the system are the RIPVServer, RIPVClient, and RIPVServices. The system is written for the Java 2 platform\(^{22}\) and takes advantage of *Remote Method Invocation* (RMI) for interacting with its distributed PEs.

![Figure 4.1: The core components of the RIPV system](image)

Functionality of this DISOP is offered to the RIPVServer in the form of a RIPVService, which *registers* itself with the server and by means of a common interface, known as the *GenericRMIIInterface*. This interface makes a set of methods available that the server is

---

\(^{20}\) Nonstandard implies that the service is not available locally on a universally recognized protocol port identifier.

\(^{21}\) Services and users may reside locally as well.

44
capable of calling to access the service. The services developed thus far include an application called the *SerialPortServer* that communicates with an ocean vehicle via a telemetry node by sending data to a serial port, a remotely accessible database called the *DatabaseServer*, and an application that can communicate with a visualization program like Matlab called the *DataVisServer*. The application communication protocol is based on *message passing* between the server, services and clients during a *session*.

Information passed to the server, called *Messages*, is dealt with in the subsystem known as the *TaskManager*. This Thread\(^{23}\) delegates responsibility after parsing either the *Header* or *Footer* of a Message that has been passed to it from one of the *assistants*. These assistants are each responsible for their own *MailBox* located in the *PostOffice*. No record of a Message is kept on the server hence it is a *stateless* server with respect to the client. The server does know that clients have contacted it, however, and also about the types of RIPVServices that have registered by keeping a *ServiceReference* that describes a service completely using a *ServiceDescriptor*.

A PostOffice delivery system is employed at the RIPVServer and RIPVServices to keep track of the 4 types of Messages: *Incoming and Outgoing ServerMessages* or *Incoming and Outgoing ServiceMessages*. The names are with respect to the part of the system that a Message is traveling to (from), so a Message that leaves a service or the server will always be of type Outgoing and vice versa. The PostOffice is responsible for scanning a

---

\(^{22}\) The name given to versions of the Java Development Kit (JDK) version 1.2 or higher [http://java.sun.com]

\(^{23}\) [http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html]
Message and placing it in the appropriate MailBox after distinguishing the type of
Message that has arrived.

MailBoxes are prioritized queues that have 3 separate priority queues for input, called
Inboxes, ranging from least important (priority 1) to most important (priority 3). A
MailBox also has an Outbox that Outgoing Messages are placed in for transport. A
Message consists of a Header and Footer. The Header contains the relevant information
about the origin of a Message as well as the desired command to be executed and a
Footer contains relevant information about where the Message has been.

A user that wishes to access the services provided by the server will download the
RIPVClient applet from the server's location and get a listing of the services available by
successfully logging on to the GUI presented by the applet. The client is then capable of
querying the services to perform tasks. To illustrate the power and flexibility of the RIPV
system, consider the following hypothetical situation:

At MIT, in the Ocean Engineering Department, a graduate student powers
up the Design Lab's Silicon Graphics Workstation Web server,
deslab.mit.edu, and starts the rmiregistry (Appendix E) and sets the user
permissions in the permit file before launching the RIPVServer on port
4444. Meanwhile, a collection of Gliders is sampling off the coast of
Bermuda and has been configured to use Freewave modems; each Glider
has a unique identification (either a serial number or name mapped to the
serial number) and the modems are connected to the serial ports of an on-
shore PC that is running the Windows 98 operating system. The
researcher in charge of this site does a one-time installation of the Java
Runtime Environment (JRE) on the PC and then launches the executable
RIPVService (called the SerialPortServer) from the JAR file that has been
sent to him as part of the software package included in the Glider Mission
Control.

At the Webb Research Corporation in Falmouth, Massachusetts, an
engineer has finished updating the SQL database that resides on a
Windows NT box with the latest information about the gliders operating in
Bermuda, and he then launches the DatabaseServer. And finally in
California, a technician at MBARI initiates his version of Matlab on a Linux machine and then launches the DataVisServer with the appropriate permissions. After this initial setup, the research team in Greece that is responsible for the Gliders in Bermuda accesses the homepage of the RIPVServer, located at http://deslab.mit.edu/~rdamus/ripv. Their Netscape browser spawns the RIPVClient applet, gets a listing of the available services from the RIPVServer. The first thing they do is upload their new binary executable control code to the RIPVServer, and query the Server for a listing of files to ensure that the file was properly transmitted. Then they successfully perform command line interfacing to the gliders, getting a directory reading of the CF1 microprocessor on board, and offload the mission logs with a reliable YMODEM transfer. Interested in the contents of these files, they ftp the files from the server to their own machine and parse the files with their own scripts. By querying the database for recent activity, they learn that 3 of the 4 gliders have not fully performed their mission objectives and thus they proceed with updating the mission parameters ASCII file and send the GO command that causes the gliders to submerge and continue. The glider that has finished its missions has returned a hefty data file that is parsed and then sent back to the RIPVServer as a 3-dimensional data set, and the DataVisServer is contacted for action. Within moments a JPEG file is returned to their RIPVClient GUI and they mull over the results, decide on a new mission for the last glider, and upload yet another binary executable to the RIPVServer. Final preparation is made with an upload of this executable binary file from the RIPVServer to the last glider and then the GO command is given. During this time, a team of scientists at URI has queried the same set of gliders and offloaded the same data files, but cannot configure the glider for a new mission because of lack of the appropriate permissions.

4.1.1 The RIPVServer

The RIPVServer is a multi-threaded server that listens for TCP/IP Socket connections from the client on a specified port, 4444. By virtue of extending\(^\text{24}\) the \texttt{java.rmi.server.UnicastRemoteObject} class, the RIPVServer is also considered a Remote Object, meaning that it has the ability to make Remote Method calls to the RIPVService. Upon initiation, the server spawns a ServerSocket Thread that listens for clients to connect to the Socket, and then binds that client by first tagging it with an

\(^{24}\) in Java, a child class will \textit{extend} a parent class to inherit the capabilities of the parent. In C++, this is known as inheriting
idBadge and then passing control over to one of its Handlers. A Handler is a Thread that becomes explicitly responsible for watching all communications over the Socket, thereby relieving the server of that task, so that it may focus its resources on registering services that establish a connection through RMI, and supporting the TaskManager’s activities. Thus the main purpose of the RIPVServer is to act as the facilitator of communication between the client and the services. Figure 4.2 is a conceptual layout of the RIPVServer and figure 4.3 shows the star network configuration of the RIPVServer and its RIPVServices.

Figure 4.2: RIPVServer conceptual layout
4.1.2 The RIPVClient

The client program, which is an extension of the `javax.swing.JApplet`\(^{26}\) class, presents the GUI to the user after being loaded in a web browser as shown in Figure 4.4 (section 2.2.3.1). The set of Java classes available in JApplet is based on Lightweight components that allow the development and drawing of modern GUI controls (section 4.2) [Horstmann 99]. The primary justification for the client program being an applet is to avoid loading resources into the memory of the client’s machine, thus allowing the freedom that the client be located anywhere in the world and still have access to the functionality provided for by services. For instance, remotely communicating with a Glider that has surfaced and needs a new command set is now possible without the operator having to be within close physical proximity of the location where the vehicle is sampling and telemetering data to.

Since an applet is a foreign program running within the runtime environment of the browser, Java has made its ability to access system level resources on the client machine almost impossible. A way around this problem is by incorporating the client-server

\(^{26}\) [http://java.sun.com/j2se/1.3/docs/api/javax/swing/JApplet.html](http://java.sun.com/j2se/1.3/docs/api/javax/swing/JApplet.html)
architecture that the RIPV employs. To specifically address the issue of providing a means for the user to upload files to the RIPV Server for distribution to services, a CGI script (see Appendix D) is presented in a window adjacent to where the applet resides in the browser (see figure 4.4). The user could always use ftp or a similar mechanism to upload the file, but included in the RIPV distribution is this useful CGI script to assist in uploading files.
Figure 4.4: RIPVClient layout with CGI upload in Web browser

RIPVClient presents the user with the GUI to access the RIPVServices via the RIPVServer.

CGI upload window uses HTML buttons to open a File Chooser dialog for uploading of files to the RIPVServer from the client’s machine.

Netscape Web browser

52
The RIPVClient first establishes a Socket connection to the RIPVServer from the web browser, and instantiates a `ClientSocketHandler` that becomes responsible for ferrying data from the GUI by listening to an `ObjectStream`\(^{27}\). This abstraction layer is very powerful and does not restrict the data type being passed between the client and server [Hughes 99]. It is suitable for presenting the RIPVServer with instances of a Message, a subclass of an `Object`\(^{28}\). In order to determine what actions should be taken based on receiving a Message, a mapping is used based on the `serviceID` tag in a Footer of a returned Message. This map is discussed further in Appendix B.

The Java language provides `Listeners` (associated with components that make up the GUI) that pay attention to user actions like mouse-clicks or button-presses. In the Java Event Model, an `ActionEvent` is a semantic event, which indicates that a component-defined action occurred [Java SUN]. Therefore the RIPVClient has several `ActionListeners` that respond accordingly by calling methods associated with the `GuiInterface` to update the GUI. The `GuiManager` is the overseer of the RIPVClient’s GUI, and is capable of updating components with results from the RIPVServer.

Clients are given unique identification numbers by the RIPVServer in order to keep track of the origin of a Message so that it returns to the proper recipient. A description of identifying the clients is presented in Appendix A.

---

\(^{27}\) [http://java.sun.com/j2se/1.3/docs/api/java/io/OutputStream.html](http://java.sun.com/j2se/1.3/docs/api/java/io/OutputStream.html)

\(^{28}\) [http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html](http://java.sun.com/j2se/1.3/docs/api/java/lang/Object.html)
4.1.3 RIPVClient’s Graphical User Interface layout

This section outlines how the GUI is arranged. The Abstract Window Toolkit (AWT)\(^29\) is Java’s library of classes that contains a number of common graphical widgets [Campione 99]. To lay out a GUI in Java, you place Components (java.awt.Component\(^30\), the superclass of the AWT) in a java.awt.Container\(^{31}\). Typically, common base components like the javax.swing.JPane\(^{32}\) are added to containers with a layout manager to create the desired interface characteristics. As previously stated, the RIPVClient uses the javax.swing package of components for GUI presentation.

The RIPV system is designed to be a dynamic interface to a changing set of services. Over the course of a session, there are a variety of scenarios that will represent the state of the system. RIPVServices might perish, or new services will introduce themselves and register with the RIPVServer with a varying degree of frequency; furthermore new services might be coded and introduced by a new service provider. There are two immediate consequences to this unpredictable behavior: the RIPVClient must have a GUI that can handle this periodic updating and removal of services, and the GUI architecture itself must be capable of providing the developer with a framework to introduce new RIPVServices.

\(^{29}\) http://developer.java.sun.com/developer/onlineTraining/awt/contents.html#AWTIntroduction

\(^{30}\) http://java.sun.com/j2se/1.3/docs/api/java/awt/Component.html

\(^{31}\) http://java.sun.com/j2se/1.3/docs/api/java/awt/Container.html

\(^{32}\) http://java.sun.com/j2se/1.3/docs/api/javax/swing/JPanel.html
The graphical widget known as the `javax.swing.JTabbedPane` nicely solves the issue of keeping track of a set of components. It allows successive components to be added to the root of the tabbed pane as panels. Each panel is given a tab that protrudes out and can be selected by the user to bring a particular tab to the forefront of a set of tabs thereby exposing the panel associated with the tab. The tab serves as a means of labeling the newly added component. This is akin to adding a card to a Rolodex and then thumbing through the cards to change the card you are observing.

![Tabbed Pane]

*Figure 4.5: The ServiceLayoutPanel and the four JPanels that comprise it*

The user interface must reflect the capabilities of the services, thus an important feature the RIPV system provides is a level of organizing content for the GUI. This framework allows for relatively easy introduction of services to the GUI and is quite modular in how to approach extending the functionality of the GUI to represent what a new RIPVService is capable of. The functions performed by each RIPVService are captured in the collection of `ServiceLayoutPanel` panels that are added to the tabbed pane. These specific panels differ in their presentation, but are implemented in a similar fashion to encourage the development of new services. Each ServiceLayoutPanel is actually a JPanel divided

---

33 http://java.sun.com/j2se/1.3/docs/api/javax/swing/JTabbedPane.html
into four quadrants (figure 4.5) that utilizes a `GridBagLayout` manager to arrange the contents of the three core panels that comprise it, the COM munications Management panel (COM panel), the File Management panel (FIM panel) and the Mailbox Management (MAM panel). The fourth panel is based on what service the ServiceLayoutPanel is responsible for. Currently, the DataBase (DAB panel) and Data Visualization (DAV panel) occupies the fourth panel.

### 4.1.3.1 The COM Panel

![The COM Panel](image)

The MailBox address of the DatabaseServer’s connection to the database.

**Status Labels:**
- Talking To [TT]
- Last Action [LA]
- Priority
- Service Status [SS]

The last command sent was a query that selected all of the data from the MISSION table.

The query was a Priority message, but not an Urgent one.

All of this activity took place because the user knew the DatabaseServer was active and ready.

Figure 4.6: The COM Panel

The COM munications Management panel is the main area in the GUI for performing command line interfacing of an RIPVService’s resource. It is always included in the GUI and spans both rows of the ServiceLayoutPanel for the SerialPortServer to allow for more room in the reply area. Figure 4.6 outlines the important parts of this panel.
4.1.3.2 The MAM Panel

Messages are passed to MailBoxes that reside on RIPVServices, thus necessitating a means of selecting an appropriate MailBox for sending commands to. The Mailbox Management panel provides this tool in the form of a `javax.swing.JTree`\textsuperscript{34} interface. Figure 4.7 depicts the MAM panel.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{mam_panel.png}
\caption{The MAM Panel}
\end{figure}

\textsuperscript{34} http://java.sun.com/j2se/1.3/docs/api/javax/swing/JTree.html
4.1.3.3 The FIM Panel

Keeping track of the files that have been uploaded from the client’s machine or that have been downloaded from an RIPVService is relegated to the File Management panel. Two buttons allow the user to either scan the contents of the upload or the download directories that are specific to the RIPVServer. See Appendix E for how to set the directories (including permissions on the directory) that the FIM panel has access to.

Figure 4.8 outlines the FIM panel.
4.1.4 The RIPVServices

The three core services each perform a basic function to extend the capabilities of a vehicle operator. Therefore, the SerialPortServer provides access to a serial port, the DatabaseServer to a database, and the DataVisServer to some type of data visualization. A RIPVService, like the RIPVServer, extends the java.rmi.UnicastRemoteObject class for RMI capabilities and can therefore be accessed like a remote object on the network after registering with the RIPVServer and passing a reference of itself (specifically, a reference of its interface) to the RIPVServer's ServiceReference. The data channel that the RIPVService employs for communicating with its underlying resource can either be based on the dynamic Handler archetype or a static connection that is established with some user-defined code. Since a Handler is a process that manages itself, the RIPVService is capable of managing multiple independent connections. Examples of dynamic resource allocation:

- A reference to a serial port
- Multiple connections to different databases
- Interfacing a Socket

Static connections include:

- Connecting to a SQL database (specific driver for the DBMS you wish to access)
- Writing to a file to pass data to another process

A naming convention for a RIPVService is outlined in appendix A.

After determining what resources the RIPVService has a handle on, MailBoxes are created in the PostOffice that correspond to the particular handle. It is a reference to these MailBoxes that is passed to the ServiceDescriptor of the RIPVServer. In order to
differentiate between services, they are given unique 6 digit identification numbers. This mapping is described in Appendix B, and takes place in the design and compilation of a new service. As a result, development of a new service must take place in concert with the adaptation of the GUI. This is a standard way of developing a distributed information architecture. It is almost impossible to avoid the coupling of the functionality presented by a service with the design of the interface to this service (i.e. the GUI). One part of the system has to know what the other parts can do. Thus, a limitation of this system is that all services that the system can support must be available in the GUI’s layout. This ensures the ability to interface a service that connects to the system. Services may register as they please, but coordination with the administrator of the RIPVServer is necessary if the new service is to function as a part of the RIPV system. 
4.1.4.1 SerialPortServer

The vast potential of exposing an interface as useful as the serial port to interaction by a remote user is harnessed in this service through a combination of RMI’s functionality and the javax.comm\textsuperscript{35} package. This package provides the low level methods necessary for accessing the TX and RX pins of the serial port. Therefore, the SerialPortServer is a fully functional serial port interface, completely capable of sending data to a Vehicle’s microprocessor and then reading the response using hardware like an RF modem (figure 4.9, figure 4.10).

![Serial Port Interface Diagram](http://java.sun.com/products/javacomm/)

Figure 4.9: Serial port interfacing using the java.comm package

A SerialPortHandler is spawned for each port that is enumerated and remains associated with the referenced port indefinitely or until an I/O error. What is connected to the serial port is up to the user. For example, interaction with an AUV or other type of vehicle requires the serial port being directly connected directly to the communications channel associated with the onboard microprocessor, like the radio modem link or serial

\[\text{http://java.sun.com/products/javacomm/}\]
communications tether. To control the rate of data exchange and insure commands are not sent when the vehicle is not ready to receive them, the SerialPortHandler queries the Carrier Detect (CD) pin of the serial port and only will communicate when a connection is established. It is possible that intermittent comms will lead to poor performance, however this is beyond the scope of this service and typically is handled in the data transfer protocol of the telemetry hardware. For file transfers, one of the first non-commercial Java implementations of YMODEM protocol is used for uploading and downloading. This protocol uses a CRC error correction scheme, keeping track of the current 16-bit CRC value by updating a CRCCITT16 object.
Figure 4.10 SerialPortServer conceptual layout
4.1.4.2 DatabaseServer

This server represents a means of connecting to a RDBMS and executing queries that pertain to the relevant data schema present in the database. The RIPV defines its own database schema that can easily be duplicated and extended to capture the relevant information associated with underwater vehicles. Appendix C shows the generic version. For perusal by the WRC Glider, the VEHICLE name of the table was changed to GLIDER. Upon establishing a connection with the database with a Java JDBC\textsuperscript{36} driver, the DatabaseServer registers with the RIPVServer and then database queries are executed and the information returned via RMI calls in the GuiInterface. The information from the data is displayed in table form in the RIPVClient’s GUI by means of a DatabasePanel.

\textsuperscript{36} Java Database Connectivity, http://java.sun.com/products/jdbc/
Figure 4.11: DatabaseServer conceptual layout
4.1.4.3 DataVisServer

Matlab was chosen as the visualization toolkit because of the power of its computational engine and scripting language. The latest version of Matlab, 6.1, allows for easy interaction with Java classes and methods through its supplied Java Virtual Machine, however, this port of Matlab did not exist when the RIPV code being laid out for the DataVisServer was first developed. As a result, the DataVisServer communicates with the Matlab computing environment in a seemingly awkward, yet straightforward manner of writing data to and reading data from predefined files. The DVS signals to Matlab that a new data file should be read and then awaits the completion of the computation that generates a jpeg file that can then be sent back to the RIPVServer and exported to the RIPVClient’s ImagePanel for inspection. The only other piece of code that needs to be running is a .m file, called ripvOut.m (see Appendix D for the code) that is launched in the Matlab working directory and runs until Matlab is shut down. The DataVisServer cannot presently wake up a sleeping copy of Matlab.

To achieve maximum flexibility in this design, the data file read by Matlab is configurable to support different visualization requirements. It is assumed that the clients will need to do their own processing of and formatting of data that has been downloaded from a vehicle. Then, the formatted data is stored in a column separated ASCII text file to support multi dimensional data and uploaded to the RIPVServer after the ‘vis:’ command is entered at the Command Line of the COM panel.
Figure 4.12: DataVisServer conceptual layout
4.2 Features

These are the design decisions that make up the RIPV tool:

- DDP framework
- Remote Method Invocation
  - Allows for updateable interfaces to remote objects on the network
- Message passing architecture
  - PostOffice delivery subsystem that uses first-in-first out (FIFO) queues for Message storage and retrieval
- Serial, pre-emptive task execution
- Priority based queues
- Multi User platform (supports Socket connections)
- Multi Threaded
- Client GUI presented in a Web browser
- Serial port interfacing
  - YMODEM file transfer at up to 115200 baud\(^{37}\)
  - Enumeration of underlying communications ports
- Visualizations presented to user via WWW
  - Powerful Matlab visualizations
- SQL database interfacing
  - Well defined schema to represent vehicle data\(^{38}\)
  - Queries returned to user GUI in table format

\(^{37}\) bits per second

\(^{38}\) Appendix C
Chapter 5

Code Architecture

5.1 Objectives

There are many issues that need to be addressed simultaneously in building a distributed system [Rajkumar 95]. The following set of objectives was addressed in composing the RIPV framework:

- The system should be configurable to a variety of implementations so as to support adaptation of and introduction of new resources.
- The system should be capable of deployment in heterogeneous computing environments.
- Interacting with resources will happen over a range of temporal scales.
- Multiple users will be accessing the system, with an emphasis on them being remotely located with respect to the device they will be interfacing.
- All system clients’ components should be loaded dynamically over the Internet. On a client, local software installation should not be necessary.

- Security
- Scalability

5.2 Design Considerations

The challenge in designing this system stemmed from the essentially “open-ended constraints” imposed by a subset of the requirements listed in the previous section. Designing a robust and adaptable system necessitated abstracting the fundamental
features desired in handling the above requirements, and spurred my interest to evaluate current versions of systems that accomplish these criteria.

5.2.1 Real Time Systems

Given that one of the immutable requirements was that this system need to present the user with a means of accessing data on an AUV or other ocean vehicle (like the Slocum glider), the temporal nature (i.e. the user will need to access the onboard file system of a robot through a telemetry link that exists during the interval of time that the vehicle is within the range of the link) of real-time system design was a natural choice for comparison of how these types of problems were being solved. Further, when an AUV or glider surfaces and requires immediate attention, the interaction will take place in a near real-time fashion. Specifically, one can characterize the interaction as soft or hard real-time [Kopetz 97], depending on how you define the consequences of missing a deadline. What this investigation elucidated was that (1) a real-time approach, although desirable on a certain level, was not practical for this system to support, and (2) that developing the concept of tasks for the system to carry out would be a useful abstraction to implement.

There are several assumptions here that eliminate the need for real time deadlines. First, vehicles have a degree of redundancy that does not lead to a failure mode if communications are intermittent. Second, given the spurious nature of communications over a vast distance (i.e. via an RF modem) another level of complexity is added to a system [Rajkumar 95] that supports establishing deadlines for milestones when there is no guarantee that the data packet being transmitted arrives. Third, the nature of common
AUV functionality like surveying, map making or other application specific missions is a process where decisions about what is the next important thing for the vehicle to do is either made by the human operator or is a function of the behaviors programmed into the control software. This does not mean that truly autonomous systems should be free of deadlines, rather the RIPV system will be a platform where investigating how to impose deadlines should and will be able to take place.

5.2.2 Telesemantics

The specific users, or clients would make requests to retrieve data that resided somewhere (on a vehicle, for instance), but the delivery of this information would have to be mediated by a central repository, or server. It became apparent that in order for this thesis to be useful to the community of individuals (like research scientists) who use instrumentation as a means of understanding their environment, incorporating the ideas associated with teleoperation, mobile networking and nomadic computing would be useful additions to the traditional model of client-server interactions in current DIS architectures like a Geographic Information Systems (GIS). NASA pioneered teleoperation, or the ability to remotely manipulate an object [NASA].

Recent experiments with Web-based telemetric and teleoperation (WTTS) systems have been attempted [Moon 99]. Implicit in this framework for thinking about delivering data to a requesting client is the observation that the utility of the system would be extended if a remote machine could perform a duty, or service, to the client-server system. Essentially this was the motivation for distributing the computation: to allow for the functionality in the system to be limited by the diversity of its services.
5.2.3 Transport Layers

Prior to actually writing any code for the RIPV system itself, a survey of the delivery methods of information over the Internet were evaluated and it was determined that the TCP/IP protocol best fit the needs of this system. As opposed to connectionless transport layers like UDP, a connection-oriented style like TCP/IP guarantees ordered packet arrivals between two nodes and provides all the reliability needed to communicate across the Internet [Comer 96]. This is the kind of performance that would be necessary if multiple clients were being hosted. Loss of packets could not be tolerated as the number of users grows, quite simply because the complexity of building redundancy should already be handled at a low level if you want to focus on the development of an adaptable system framework.

An attractive alternative that is similar to UDP is multicast, a datagram network protocol that allows an application to place a single packet on a network and have that packet transported to multiple recipients [Hughes 99]. The largest deployment of multicast on the Internet is the MBone, which is an experimental multicast framework that transports multicast packets through TCP/IP tunnels between multicast enabled islands. Currently, the overhead associated with having routers determine whether adjacent networks support multicasting undermines the utility of this delivery method. Perhaps in the future, the barriers to wide scale deployment of multicast packet transfers will diminish and it will be a more desirable form for implementing multiple client systems.
So with the decision to implement TCP came the requirement that a Socket would be used for streaming data between the client and the server, as this is Java's representation of a TCP connection. On the service side, RMI was the choice to solve the problem of handling the Socket connections and keeping tabs of remote services as described in Chapter 3. Figure 5-1 shows the socket connection between the RIPVClient and the RIPVServer that is used for transmitting information between them.

![Socket connection between the RIPVServer and RIPVClient](image)

**Figure 5.1: Socket connection between the RIPVServer and RIPVClient**

### 5.2.4 Application-Oriented Design

The *application-oriented* approach taken in designing this distributed system is considered such because the underlying RIPV framework represents a conventional application program solution to a problem that is doled out to PEs. A variety of factors lead to this design procedure, rather than having built the system around a communications protocol as the driving design criteria. Since it was envisioned that this software would operate within the context of existing robot systems, it is intended to be useful over a broad range of underwater robotic systems that already are in use, but can be molded to accommodate future incarnations as well.

As an example as to how the application model drove the solution, consider the model in figure 5-1. It lacks completeness if there is more than a single user of the system, thus there would need to be a means of supporting *multiple-to-one* interfacing, either from a client-to-server or a device-to-service perspective. The concept of a Handler, or process
that manages an interaction, was introduced. Implemented as a thread
`(java.lang.Thread)`, a handler could be used on the server side for watching a socket
connection and then it would be known as a `SocketHandler`, or it could be used on the
service side for paying attention to a serial port and be referred to as a `SerialPortHandler`.
Therefore, the RIPVServer is considered a *concurrent* server because it can handle
multiple requests at a time, whereas a RIPVService that uses a Handler is concurrent with
respect to the devices that it can concurrently interface. This leads to the question of how
to handle requests, which is covered in the next section, 5.2.5.

This aspect of concurrency is implemented at a certain cost of the actual semantics of
addressing multiple remote executions, however, and therefore a variation on a strategy
known as *mutual exclusion* is borrowed from Sun Microsystem’s Open Network
Computing (ONC) RPC. The ONC RPC mechanism specifies that at most one remote
procedure in a remote program can be invoked at a given time [Comer 96]. Similarly, the
`RIPV` communications model provides for single event task execution on the remote
services through extensive use of Java’s *synchronization* techniques. For instance, one
does not have to worry about a procedure like doing a directory read of a glider’s
microprocessor being interrupted. Unfortunately, all of this does not guarantee reliability
in the strict sense. Requests may be lost, and connections intermittent, especially when
communicating with a vehicle over a vast distance. Thus, the `RIPV` system is a *zero-or-
more* system, meaning that the client may not receive a reply from a request. The system
does keep track of active connections and makes this information available to the user
(see figure 4.6).
The RIPV system is designed to act as an interface layer on top of pre-existing hardware solutions. The system is minimally invasive in this sense and behaves like a command line interface to the underlying resource, be it a vehicle attached to a serial port, a connection to a database, or a visualization package like Matlab. Imposing a protocol to drive communications between the RIPVService (i.e., the SerialPortServer that is watching a serial port attached to a vehicle) and its data device was done loosely. As opposed to developing an infrastructure from the bottom most layers like the hardware instruction set used to power the operating system of the controlling microprocessor, mutual exclusion guarantees the entire conversation between a service and its respective device takes place based on the act of presenting the device with a request and awaiting a reply.

Explicit guidelines for how to interface different resources are adopted from rules on how an operator would normally interface the data device. Support for an interface definition language, such as the one in the CORBA framework, was deemed unnecessary because of the more general GenericRMIInterface used to communicate between the server and services. It is this interface that gets modified by a developer to represent the new method calls to remote services. Thus, the programmer who develops the capabilities of the RIPV system must be careful when implementing a service and be consistent that a method defined in the remote interface is reflected in the service itself. On the low level, application-specific code must be written by the programmer to achieve desired functionality at the service, but this would have to take place regardless.
5.2.5 Communication Model

The communication model was born out of a need to send information from one part of the system to the other, and have a response (that arrives at the appropriate recipient) generated. Information from the client, server, or services is encapsulated in an object known as a *Message*, which is passed between modules internally from storage *MailBoxes* that reside in a *PostOffice*. This necessitated identifying the components of the system that would be producing information (i.e. the services have unique service id’s) as well as keeping tabs on where a message originated (i.e. from what client id). *Mappings* are used to tag a service uniquely, as well as to define peripheral functionality that the GUI might be interested in. An example would be the ability to request the contents of the uploaded file directory. Since this involves updating the GUI, the handler responsible for the client’s socket (the *ClientSocketHandler*) examines a returned message by checking the service id and then calling the appropriate method to write to the GUI from the *GUIInterface*. See Appendix B for a description of the implemented service IDs.

The services make themselves known to the RIPVServer by following a *registration* scheme whereby they pass a reference of their remote interface to the *ServiceReference* objects that are created when they register via an RMI call to the RIPVServer’s *GenericRMIIInterface*. This reference allows the RIPVServer to communicate via an RMI link that is transparent to the client. The default is for a Thread on the client to periodically check the RIPVServer for new services that have registered; the user can request this information explicitly at any given time.
The activities of the RIPVServer and RIPVServices are triggered asynchronously. User input like command line interfacing a vehicle, or receiving a graphical output from Matlab all happen over different time scales, and it is important that the system be able to store actions that need to transpire once requested. An event channel is adopted that is based on priority queues which is similar to [Moon 99], but not in a real time sense. A storage device called a MailBox contains four separate queues for handling messages; three are prioritized and relate to incoming mail, referred to as an IncomingServerMessage or IncomingServiceMessage (ISM), and the fourth queue stores outgoing mail known as an OutgoingServerMessage or OutgoingServiceMessage (OSM). These queues are implemented as First In First Out (FIFO) and depicted in figure 5.2.

![MailBox queues diagram](image)

**Figure 5.2: MailBox queues**

5.2.6 Process Management

The message delivery system and inter process management is overseen by a TaskManager. A TaskManager’s primary functionality is to serve as the message parser to determine what action needs to be taken by the system to accomplish the command in the body of the message Header. To do this, the TaskManager is passed messages from either the InboxAssistantThread or OutboxAssistantThread, which are a pair of threads.
assigned to each MailBox that is created to store messages for the server or service. For convenience, the server has only one MailBox, called the “RIPV Post Office”, and this is where requests from the client and replies from the services are stored. A service, on the other hand, can instantiate as many MailBoxes as it needs to keep track of messages for different handlers or for coping with requests about its status. By default, a TaskManager watches over whatever MailBox it has been assigned to, and only is concerned with that MailBox. The nature of the service using a MailBox dictates what other objects the TaskManager has access to. For instance, on the SerialPortServer, a SerialPortHandler watches over a specific serial port, and thus in order to place the command from a message on the TX pin of the serial port, the TaskManager has a reference to the SerialPortHandler so it can access the appropriate communication stream.

Figure 5.3 and 5.4 shows the layout of the RIPVServer and RIPVServices, respectively. These architectures are comprised of the same components that function in different capacities based on what module (server or service) of the RIPV system they are in. Each server or service contains one PostOffice. Common to both implementations is how an incoming message is received and then scanned and placed in the appropriate MailBox in the PostOffice based on the mailboxAddress variable in its Header. Each MailBox has a MailBoxKey that can lock the Mailbox when a command is actually being executed and this is how the mutual exclusion is supported.
Figure 5.3: RIPVServer architecture with pseudo code
parseHeader(message) {
  if (usingAHandler) {
    search for handler in handlerV Vector
    case handlerExists:
      send command from header.body to handler
    default:
      alert RIPVServer to failure
  } else {
    header.getBody()
    perform action with command. If action relates to the
data device this service watches,
    pass control over to the data device.
    Or return a new OutgoingServiceMessage
    with results of completed action
  }
}

parseFooter(message) {
  call back to the RIPVServer with result
  using serviceReply() method
}
The InboxAssistantThread’s algorithm is a serial, *pre-emptive* searching pattern that guarantees that highest priority messages are removed from their queues first. Once an incoming message is removed from an Inbox queue, the TaskManager parses its header. In the RIPVServer, the message is either ferried off to a service via an RMI call from the ServiceReference’s GenericRMIIInterface, or the TaskManager handles the message locally and the reply is placed in the Outbox of the RIPV Post Office. An OutboxAssistantThread delivers messages off of the Outbox queue back to the TaskManager for parsing of the Footer. The TaskManager has a reference to the SocketHandler and returns the outgoing message to the RIPVClient for perusal by the ClientSocketHandler and GuiManager of the RIPVClient.

The RIPVService operates in a similar fashion to the RIPVServer, only instead of a single MailBox in the PostOffice, there are at least as many MailBoxes as there are connections to different data devices. The PostOffice scans an incoming service message that is received via an RMI call from the RIPVServer and places the message in the appropriate MailBox. Again, an InboxAssistantThread passes this message to the TaskManager whose parses the Header, only instead of checking a ServiceReference an action is taken based on the contents of the Header’s body. This action may constitute passing a query to a database via a reference to the database’s connection, or writing information to a file so that Matlab loads the contents of the data file for imaging, or perhaps it is to put a command on a SerialPortHandler’s serial port reference to a vehicle’s telemetry hardware. A response is returned to the appropriate MailBox’s Outbox queue and since an OutboxAssistant is watching this, the message is passed to the TaskManager to parse the Footer. A successful reply from the RIPVService to the
RIPVServer is finalized with an RMI call to the `serviceReply()` method in the GenericRMIInterface of the RIPVService’s reference to the RIPVServer.

The majority of the RIPVClient’s processes are handled by Java’s event processing architecture implicit in the GUI widgets and applet class, thus a TaskManager is not employed on the client side to dictate the actions taken by the RIPVClient. Instead, a `ClientSocketHandler` mediates the message passing through an `ObjectStream` that is layered on top of the socket connection between the RIPVClient and RIPVServer. The ClientSocketHandler processes messages and affects change in the GUI by calling methods that are made public in the `GuiManager`’s interface, `GuiInterface`. These method calls to the GUI components contain the new information that has been returned by the RIPVServer and the user is interested in. Figure 5.5 illustrates this chain of commands.

![Diagram](image)

**Figure 5.5: RIPVClient process management**

### 5.3 Package Structure and Class Hierarchy

The classes that comprise the complete RIPV system are grouped into packages by functionality; the root package is referred to as `mit.oe.deslab.rdamus.ripv`. In Java, the
super class of all classes is the java.lang.Object. All objects created by the programmer inherit from this class. For the RIPV system, the grouping of class behaviors into packages is used as a means of organization for clarifying the role of a particular piece of the system, with the core group of classes (the RIPVServer, RIPVServices and RIPVClient) belonging to the root package. A complete listing of the classes and the tree structure associated with them is included in Appendix F.

<table>
<thead>
<tr>
<th>Package</th>
<th>Subpackage</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit.oe.deslab.rdamus.ripv</td>
<td></td>
</tr>
<tr>
<td>mit.oe.deslab.rdamus.ripv.assistants</td>
<td>mit.oe.deslab.rdamus.ripv.exceptions</td>
</tr>
<tr>
<td>mit.oe.deslab.rdamus.ripv.handlers</td>
<td>mit.oe.deslab.rdamus.ripv.managers</td>
</tr>
<tr>
<td>mit.oe.deslab.rdamus.ripv.messages</td>
<td>mit.oe.deslab.rdamus.ripv.panels</td>
</tr>
<tr>
<td>mit.oe.deslab.rdamus.ripv.services</td>
<td>mit.oe.deslab.rdamus.ripv.trees</td>
</tr>
</tbody>
</table>

Table 5.1: RIPV packages

5.4 Database Schema

A relational database model was developed for the purpose of storing information that is extracted from an underwater vehicle so that it can be accessed at a later date for interrogation. The requirements for underwater vehicle operation were analyzed and the conceptual schema that resulted from this analysis is presented in figure 5-6. The components of this schema are individual tables containing relations that are pertinent to characterizing underwater vehicle missions and the data that is collected on the mission by individual sensors. Each relation (Vehicle, Vehicle_Data, Mission, Leg, Sensor, Sensor_Data) has attributes, one of which identifies the relation instance, or tuple,

---

39 See chapter 4 for a description of classes.
uniquely and is referred to as the Primary Key, denoted by the [PK] next to the attribute name. Each attribute is of a specific type, which is listed after the colon. A Foreign Key [FK], which references the primary key of another relation, ensures referential integrity [Elmasri 94]. A primary key is a unique identifying characteristic of the table’s relation and therefore can be referenced by a foreign key in another table to map the information from the primary key’s table to that of the foreign key. This database schema is written in SQL and supports implementation in a Database Management System (DBMS).

Figure 5.6: Conceptual Schema for database storage of vehicle data
Chapter 6

Implementation

For a complete installation and execution guide, please see the setup.readme reference located in Appendix E.

6.1 Hardware Setup

Platform independence of the code is a function of the Java language. As a result, this software functions across heterogeneous environments without hesitation. Thus far, all of the machines used to test the code have been at least Pentium class systems with more than 16MB of RAM. The Java Runtime Environment currently takes up about 5MB of hard drive space and is a one-time installation required for a hardware platforms to host Java applications to garner the JVM for interpretation of compiled code as described in chapter 3.

6.2 O/S Platforms

The requirements for implementation of the software vary based on what part of the RIPV system you plan on executing. The version of the JVM used to compile the current version of the code is Java 2, version 1.2.2. All machines that want to participate in the RIPV system must have an Internet connection and be addressable by a unique IP address. Currently there is no support for Firewalls or Proxies, although RMI does make provisions for it.
For service providers that will be hosting RIPVServices, platforms that have been successfully used include Windows NT, 2000, 98 and 95, SGI running Irix 6.x, and Solaris. If you plan on implementing the RIPVServer, your system must be able to open Socket connections. This implies that an operating system that supports server applications like Irix, Solaris, Linux or Windows Server (both NT and 2000) should be used for convenience. However, non-commercial applications like Tomcat, or the myriad of other server software can be used to make a non-server machine with an IP address capable of hosting clients. If you plan on being a client, then see the next section.

6.3 Browser Software

The RIPVClient is loaded as an applet, and therefore the browser the client that uses to load software must support the JVM that the RIPV was written in. Hot Java or Netscape and Internet Explorer versions 4.x –5.5 will most likely need to download the latest virtual machines at [JVM Microsoft, Netscape], and have the latest Java plugin for the browser.

6.4 TestBed

The design of the RIPV tool is based primarily on working with the WRC Glider, Slocum, which uses a Persistor Instruments [Persistor] CF1 32bit-microcontroller as the onboard “brain” of the vehicle. The RIPV tool anticipates interfacing this processor by incorporating thejavax.comm. as explained in chapter 4. Both a direct connection to the processor and communicating through a Free Wave RF modem has been tested.

40 http://apache.Jakarta.org
41 http://java.sun.com/products/plugin/
6.5 Testing

Testing has taken place over the course of development of the software system. The results are presented here.

6.5.1 Serial Port Interfacing

Successful tests between a CF1 hardwired to a serial port (see figure 6.1) as well as over a radio telemetry link have been conducted. The CF1 responds to command line arguments as expected (e.g. dir, del) and has been tried at 9600 baud and 115200 baud. Furthermore, an adaptation of the YMODEM protocol allows for uploading and downloading of files to the CF1. Any file size can be transferred.

![Command Management](image)

*Figure 6.1: Successful ‘dir’ command returns directory listing from CF1*
6.5.2 Database Connectivity

Queries on data stored in the Cloudscape [Java] RDBMS, using the SQL database schema shown in Appendix C have been performed and successfully returned to the RIPVClient. Figure 6.2 shows the tabbed pane that houses the returned database queries. The tab shows the text from the query and a `javax.swing.JTable` presents the user with the results of the query.

![Database Management](image)

*Figure 6.2: Returned database queries*

---

http://java.sun.com/j2se/1.3/docs/api/javax/swing/JTable.html
6.5.3 Data Visualization

Visualizations in Matlab have been processed and sent back to the RIPVClient. The visualizations are stored as high quality jpeg files and can be downloaded from the RIPVServer in addition to observing them directly in the GUI. Figure 6.3 is an example of a 2 dimensional data plot with false data to demonstrate the data visualization concept. This graph does not have a legend because it was not specified in the data file.

Figure 6.3: Simple 2-D data plot
Chapter 7

Conclusions and Future Work

7.1 Conclusions

The \textit{RIPV} tool is one of the first architectures to combine Remote Method Invocation, Java’s object oriented capabilities, a client-server model, the features of an applet, and the Web to present functionality to users that can reside anywhere on a network and still gain access to the resources they are interested in. RMI is implemented as the communication protocol for distributed computing access to remote resources and the Web is utilized as a means of presentation. This combination of interfacing and accessibility create a data information system without the need for the client to download any software to their system. This fact alone separates this system from previous implementations of client-server architectures that are platform specific in nature or might require the user to download executable code to run on their machine.

The advent of the hosting of program logic in Web space is just beginning to gain momentum in the distributed information community. The majority of Web applications are still static representations of data, without provisions for adapting the code to handle different situations. By presenting the features of the RIPVServer in an applet to the user, modern GUI controls are used to simplify interaction with the RIPVServices, a secure environment to communicate with the RIPVServer is ensured, and a broad scope of functionality can be achieved through high level application logic that is conveniently available in an applet, despite running in a Web browser with limited memory resources.
The RIPV system answers a crucial problem in information dissemination to a remotely located data device. One of the large hurdles encountered in not downloading an application to the client machine is the loss of control over System level command like enumerating files in a directory. By coupling the utility of the upload.cgi script with the logic of the RIPVServer, a simple and powerful way around the problem of uploading a file from the client to an AUV is solved.

The largest contribution of this work is the achievement of cross-platform independence for the purpose of Remote observation of an ocean resource, like an AUV or other data logging equipment that use microprocessors. There are discussions in the literature about the utility of having all of the components of this system, but no prior attempts at homogenizing the architecture for such a system have occurred. Specific solutions that require unique configurations of software fall short of providing the flexibility that the RIPV system enables. The advent of RMI has enabled Java-based programming to establish its preeminence in becoming a suitable platform to develop networked solutions to a distributed information problem such as the one addressed in this thesis.

Taking the client-server paradigm a step further, the RIPV approach presents a user with the three core functionalities provided for in the RIPVServices. These services extend the capabilities of a researcher and provide a way to develop efficient mission plans by presenting a coherent means of organization of ocean-related data.

The system can handle multiple users concurrently accessing data in near real-time, and can be scaled to incorporate more services. The system is adaptable and can be reconfigured with a moderate understanding of object-oriented programming in a
relatively short time frame to accommodate changes required by a specific application.
Methods are defined in interfaces that dictate the characteristics of the server and services, as well as the client. Thus, each fundamental part of the RIPV system can be modified, and through the use of Mappings, maintain a level of consistency to guarantee desired behaviors.

<table>
<thead>
<tr>
<th><strong>Pros</strong></th>
<th><strong>Cons</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptable</td>
<td>The clientID/serviceID requires a mapping</td>
</tr>
<tr>
<td>Integrate across heterogeneous platforms</td>
<td>Perhaps not as easily configurable as the CORBA broker architecture</td>
</tr>
<tr>
<td>Uses remote method invocation</td>
<td>MailBoxes only have 3 queues</td>
</tr>
<tr>
<td>Distributed system</td>
<td>Multiple threads can tax a processor</td>
</tr>
<tr>
<td>GUI defines the interface</td>
<td>Basically, design decisions had to stop at being completely ubiquitous.</td>
</tr>
<tr>
<td>Message passing and the ability to define new fields in the Header and Footer make for highly adaptable communications-oriented design</td>
<td>Could not write every method that can anticipate what the ultimate requirements of every project would be.</td>
</tr>
<tr>
<td>Multi threaded environment</td>
<td>Perhaps compartmentalizing of more interfaces that were specific to different classes would be a more intuitive way of organizing the methods available</td>
</tr>
<tr>
<td>Priority based queues</td>
<td></td>
</tr>
<tr>
<td>Serial port interfacing made easy</td>
<td></td>
</tr>
<tr>
<td>Client loads in a web browser</td>
<td></td>
</tr>
<tr>
<td>Remote interface</td>
<td></td>
</tr>
<tr>
<td>Gui Interface</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Advantages and Disadvantages of the RIPV System
7.2 Future Work

The work presented in this thesis should be viewed as a framework for how to design a multi-user environment for ocean vehicles with RMI services in Java. Future work with this software architecture should include extending the capabilities of the system by writing new services to present to the user. It is desirable to establish field operations with the Slocum glider to test a coherent system for coastal zone observation.

7.2.1 New Functionality

The most intuitive way to improve the capabilities of the \textit{RIPV} system is to write more services. To do this, a developer would extend the RIPVService class, decide on either the Handler or static approach for interfacing whatever new resource or application the service encapsulates, create a new unique serviceID that corresponds to this new set of actions, and then start the service. A new panel for the GUI will likely be introduced so that the functionality desired can be presented to the user. Therefore, the GuiManager and RIPVClient will have to be updated accordingly.

This research was motivated by the concept of interfacing a microprocessor and making this interface available to multiple public recipients. With that in mind, it would be useful to see a service developed that allows for true global interconnectivity with a mobile platform like an AUV. The supposition here is that the AUV has telemetry hardware that affords it a large coverage area, like a satellite card.

The RIPV system would be a useful platform to experiment with algorithm development for AUV related activities. One such activity might include improving on navigation
related objectives like path planning and use Graph Theory to help an operator plan an optimal mission route much like a traveling salesman problem is approached and solved.

7.2.2 Java Beans

Java has developed its concept of distributed applications into an Enterprise solution based on the RMI framework. Java Beans are objects that reside in a container located on a network and have special methods for exposing their state and can be used for distributed tasks. It would not be a far cry to improve upon the RIPV model by creating the services as Beans as much of the code can be used to retain the functionality. The end result would be a highly modular system with specific entities (the Beans) that could then be used as the basis for commercializing the access to data and creating new business models for ocean technology companies that provide services to researchers and scientists. See [Java Beans intro] for more information.

7.2.3 Mission Planning Efficiency

With all of the functionality that the RIPV tool provides to a vehicle operator, it would be interesting to study whether mission planning becomes an easier task. AUVs are becoming increasingly prevalent and one of the benchmarks that have set the successful incarnations apart from the rest is the ease in which a mission can be carried out. Mission planning can be a tedious undertaking, as the criteria for evaluating the success of the mission is inherently subjective and ultimately related to the scientist’s (or experiment’s) goals. Whether or not this is a limiting factor in the design of an AUV has yet to be determined; most AUVs are primarily scientific instruments and are most likely
instrumented by the engineers that had something to do with the development of the vehicle.

Controlled experiments with specific goals could be used to test an operator’s ability to cycle through a set of mission objectives and then develop hypotheses based on the data that their vehicle has returned. Performing multiple missions in succession in a given time frame would be the goal. The quality of the hypotheses and ability to finish all of the missions in the allotted time would give a heuristic means of analyzing the efficacy of the RIPV tool in mission planning.

7.2.4 Autonomous Mission Planning

As we get closer to having adaptive sampling networks in the ocean, the issue of how to plan and execute missions autonomously is raised. Currently while working at Webb Research, I am developing an extension of the RIPV system that will interact with a Slocum glider without the assistance of an operator. A user inputs a WorkFlow, which is a collection of actions that the vehicle should execute after having surfaced and made contact with the shore node. This has required the addition of a new Manager class object that keeps track of a vehicle when it surfaces, and the addition of MailBoxes to store the WorkFlows being passed to the glider. The rest of the functionality is leveraged off of the RIPV system. The RIPV framework is proving itself to be a worthy test-bed where methodologies for autonomous mission planning can be ascertained.
Appendix A – Naming Conventions

I. The package name for the RIPV system is the prefix to any class that is used in the software that is not part of the Java 2 standard packages. There are 9 total packages, including the root package.
The fully qualified package names begin with the root:

```java
mit.oe.deslab.rdamus.ripv
```

The remaining 8 package names are

- `mit.oe.deslab.rdamus.ripv.assistants`
- `mit.oe.deslab.rdamus.ripv.exceptions`
- `mit.oe.deslab.rdamus.ripv.handlers`
- `mit.oe.deslab.rdamus.ripv.managers`
- `mit.oe.deslab.rdamus.ripv.messages`
- `mit.oe.deslab.rdamus.ripv.panels`
- `mit.oe.deslab.rdamus.ripv.services`
- `mit.oe.deslab.rdamus.ripv.trees`

Thus, the addition of a new Manager type would take on the name:

```java
mit.oe.deslab.rdamus.ripv.managers.NewManager.java
```

Notes:
The fully qualified name of a class corresponds to the directory structure that should be used to store the file by replacing the ' . ' with a ' / ' (on a Windows environment). Therefore, the NewManager.java file would be found at:

```
C:\src\mit\oe\deslab\rdamus\ripv\managers\NewManager.java
```

Assuming the directory, C:\src, is where the source files are stored. See more in Appendix E.

II. Defining a RIPVServiceName to define a source (.java) file:

```java
Service Component Server
```

e.g. The service that watches the Serial Port is:

SerialPortServer

III. The fully qualified name of a service that is registered with the RIPV system is:

```java
RIPVServiceName >::< IP address
```

e.g. the DatabaseServer located at deslab.mit.edu would be called:

```
DatabaseServer>::<deslab.mit.edu
```

IV. A MailBox is characterized by the fully qualified name of a service and a communications handle:

```java
RIPVServiceName >::< IP address > COM handle Index
```

e.g. the MailBox that is associated with the COM port 1 of the SerialPortServer being run on imvros.mit.edu (an NT machine) is:

```
SerialPortServer>::<imvros.mit.edu>COM1
```

Notes:
The DEFAULT mailbox handle for a Windows platform is COM. The DEFAULT mailbox handle for a Unix platform is TTY. Thus, the MailBox for the DataVisServer located at imvros.mit.edu is:

```
DataVisServer>::<imvros.mit.edu>COM1
```
Appendix B - Mappings

IDs
A six digit number string is used to define the different aspects of the system that require uniqueness.

Service IDs
Although the RIPVServer knows about the RIPVServices because of the ServiceReference's ServiceDescriptor object that keeps track of pertinent service information like name and location, the server is clueless about the origin of RMI calls from the services. Thus, the RIPVServer can easily send a message to the appropriate service by knowing the serviceDestination and mailboxAddress parameters of the Header, but cannot discern where a returned OutgoingServiceMessage has originated from. To solve this problem, a unique serviceID(SID) is associated with each type of service. This is very similar to the concept of a process ID in an operating system platform like Unix.

<table>
<thead>
<tr>
<th>RIPVService</th>
<th>SID</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatabaseServer</td>
<td>666666</td>
</tr>
<tr>
<td>SerialPortServer</td>
<td>888888</td>
</tr>
<tr>
<td>DataVisServer</td>
<td>999999</td>
</tr>
</tbody>
</table>

Functionality IDs
The RIPVServer and RIPVServices can be sent commands that begin with the question mark, '?', character. This set of commands is useful for interrogating the server and services for information that appears in the GUI. It is possible to add more functionality by creating a larger set of these commands. The command set is shown in the table below. Each command maps to a unique functionality ID (FID) so that the ClientSocketHandler calls the appropriate method from the GUIInterface based on what the FID of the reply is. The FID is sent back as the serviceID variable of the Footer received.

<table>
<thead>
<tr>
<th>Command</th>
<th>FID</th>
</tr>
</thead>
<tbody>
<tr>
<td>?dldir</td>
<td>444444</td>
</tr>
<tr>
<td>?uldir</td>
<td>444445</td>
</tr>
<tr>
<td>?dlfiles</td>
<td>444454</td>
</tr>
<tr>
<td>?ulfles</td>
<td>444455</td>
</tr>
<tr>
<td>?mailboxes</td>
<td>555555</td>
</tr>
<tr>
<td>?status</td>
<td>555666</td>
</tr>
<tr>
<td>?services</td>
<td>777777</td>
</tr>
</tbody>
</table>

Functionality
The following table is a reference for the task associated with each command.

<table>
<thead>
<tr>
<th>Command</th>
<th>Action taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>?dldir</td>
<td>The directory used by the RIPVServer to store downloaded files from RIPVServices is returned to the RIPVCClient's GuiManager and added to the FIM panel as the downloaded directory root node of the FileTree.</td>
</tr>
<tr>
<td>?uldir</td>
<td>The directory used by the RIPVServer to store uploaded files from the RIPVClient is returned to the RIPVCClient's GuiManager and added to the FIM panel as the uploaded directory root node of the FileTree.</td>
</tr>
</tbody>
</table>
directory root node of the FileTree

`?dlfiles` retrieves a listing of the files that have been downloaded from RIPVServices to be listed in the FIM panel under the downloaded directory root node from the `?dl dir` command.

`?ulf files` retrieves a listing of the files that have been uploaded from the RIPVClient to be listed in the FIM panel under the uploaded directory root node from the `?uld dir` command.

`?mailboxes` retrieves a listing of what MailBoxes are available at the registered services, this command is issued to the RIPVServer and the results are passed back to the RIPVClient's GuiManager and placed in the MAM panel's MailboxTree.

When the user selects a MailBox, this command is issued to the RIPVServer which returns the MailBox's status to the RIPVClient's GuiManager and updates the SS label in the COM panel to reflect the status: `available` or `not available`.

The semantics for determining if a particular MailBox is available is specific to the service.

`?status` when a user is validated, this command is sent to the RIPVServer to get a listing of what services are available. The listing is passed back to the RIPVClient's GuiManager and placed in the SEL panel's ServiceTree.
Appendix C – Database Schema and Code

MISSION

<table>
<thead>
<tr>
<th>MISSION_ID: NUMBER [PK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEHICLE_ID: NUMBER [FK]</td>
</tr>
<tr>
<td>NAME: VARCHAR2</td>
</tr>
<tr>
<td>DESCRIPTION: VARCHAR2</td>
</tr>
<tr>
<td>START_T: DATE</td>
</tr>
<tr>
<td>END_T: DATE</td>
</tr>
</tbody>
</table>

VEHICLE

<table>
<thead>
<tr>
<th>VEHICLE_ID: NUMBER [PK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME: VARCHAR2</td>
</tr>
<tr>
<td>DESCRIPTION: VARCHAR2</td>
</tr>
</tbody>
</table>

LEG

<table>
<thead>
<tr>
<th>LEG_ID: NUMBER [PK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MISSION_ID: NUMBER [FK]</td>
</tr>
<tr>
<td>NAME: VARCHAR2</td>
</tr>
<tr>
<td>DESCRIPTION: VARCHAR2</td>
</tr>
<tr>
<td>START_T: DATE</td>
</tr>
<tr>
<td>END_T: DATE</td>
</tr>
</tbody>
</table>

SENSOR DATA

<table>
<thead>
<tr>
<th>LEG_ID: NUMBER [PK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENSOR_ID: NUMBER [FK]</td>
</tr>
<tr>
<td>T: DATE</td>
</tr>
<tr>
<td>VALUE: NUMBER</td>
</tr>
</tbody>
</table>

VEHICLE DATA

<table>
<thead>
<tr>
<th>LEG_ID: NUMBER [FK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>X: NUMBER</td>
</tr>
<tr>
<td>Y: NUMBER</td>
</tr>
<tr>
<td>T: DATE</td>
</tr>
<tr>
<td>PITCH: NUMBER</td>
</tr>
<tr>
<td>ROLL: NUMBER</td>
</tr>
</tbody>
</table>

SENSOR

<table>
<thead>
<tr>
<th>SENSOR_ID: NUMBER [PK]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME: VARCHAR2</td>
</tr>
<tr>
<td>DESCRIPTION: VARCHAR2</td>
</tr>
<tr>
<td>UNITS: VARCHAR2</td>
</tr>
<tr>
<td>RAW_UNITS: VARCHAR2</td>
</tr>
</tbody>
</table>

PK - denotes the Primary Key, or unique identifier, for referencing this table
FK - denotes the Foreign Key that refers to the PK of a related table
The following SQL code can be loaded into a database to take advantage of the database schema described on the previous page. The strength of this schema stems from its simplicity. It can easily be adapted to support various vehicle parameters and sensor types.

```sql
CREATE TABLE vehicle (  
    vehicle_id INTEGER CONSTRAINT vehicle_pk PRIMARY KEY,  
    name VARCHAR2(64) CONSTRAINT vehicle_name_nn NOT NULL,  
    description VARCHAR2(256));

CREATE TABLE mission (  
    mission_id INTEGER CONSTRAINT mission_pk PRIMARY KEY,  
    vehicle_id INTEGER CONSTRAINT mission_vehicle_fk REFERENCES vehicle(vehicle_id),  
    name VARCHAR2(64) CONSTRAINT mission_name_nn NOT NULL,  
    description VARCHAR2(256),  
    start_t DATE,  
    end_t DATE);

CREATE TABLE leg (  
    leg_id INTEGER CONSTRAINT leg_pk PRIMARY KEY,  
    mission_id INTEGER CONSTRAINT leg_mission_fk REFERENCES mission(mission_id),  
    name VARCHAR2(64) CONSTRAINT leg_name_nn NOT NULL,  
    description VARCHAR2(256),  
    start_t DATE,  
    end_t DATE);
CREATE TABLE vehicle_data (  
    leg_id INTEGER CONSTRAINT vehicle_data_leg_fk REFERENCES leg(leg_id),  
    t DATE CONSTRAINT vehicle_data_t_nn NOT NULL,  
    x INTEGER CONSTRAINT vehicle_data_x_nn NOT NULL,  
    y INTEGER CONSTRAINT vehicle_data_y_nn NOT NULL  
    pitch DOUBLE PRECISION CONSTRAINT vehicle_data_pitch_nn NOT NULL,  
    roll DOUBLE PRECISION CONSTRAINT vehicle_data_roll_nn NOT NULL);  

CREATE TABLE sensor (  
    sensor_id INTEGER CONSTRAINT sensor_pk PRIMARY KEY,  
    name VARCHAR2(64) CONSTRAINT sensor_name_nn NOT NULL,  
    description VARCHAR2(256),  
    units VARCHAR2(8) CONSTRAINT sensor_units_nn NOT NULL,  
    raw_units VARCHAR2(8) CONSTRAINT sensor_units_nn NOT NULL);  

CREATE TABLE sensor_data (  
    leg_id INTEGER CONSTRAINT sensor_data_leg_fk REFERENCES leg(leg_id),  
    sensor_id INTEGER CONSTRAINT sensor_data_sensor_fk REFERENCES sensor(sensor_id),  
    t DATE CONSTRAINT sensor_data_t_nn NOT NULL,  
    value DOUBLE PRECISION CONSTRAINT sensor_data_value_nn NOT NULL,  
    raw_value DOUBLE PRECISION CONSTRAINT sensor_data_raw_value_nn NOT NULL);
Appendix D – CGI Script & Matlab file

#!/usr/sbin/perl
#
# Program Specific Quickie Notes:
# * Make Sure The First Line Is Pointing To The Correct Location Of Perl 5.
# * Make Sure This Program is chmodded with the permissions '755'.
#
# Version: Time Stamp: History:
# #
# Configurable Options Follow:
#
BEGIN {

    $SAVE_DIRECTORY = "/tmp_mnt/usr2/deslabl4/rdamus/serialserver/uploaded_files";
    
    # Variable:
    # $SAVE_DIRECTORY
    # Function:
    # Defines the path to the directory on the server that should be used as the folder to save files into.
    
    # Directory Permissions:
    # a+rwx
    # Additional Notes:
    # This path should not have a trailing forward slash. Also remember that this is a path, not a URL. Use something similar to: /
    # /home/myself/www/uploads
    
    $MAXIMUM_UPLOAD = 0;
    
    # Variable:
    # $MAXIMUM_UPLOAD
    # Function:
    # Defines the number of bytes that can be uploaded. Files that exceed this limit will not be saved on the server.
    
    # Additional Notes:
    # Set this to zero in order to

102
disable size checking.

$ALLOW_INDEX = 1;

# -- Variable:
# $ALLOW_INDEX
# -- Function:
# If set to zero, files whose names begin with the word index will not be saved.
# Set to one to allow files named index* to be uploaded.

$SUCCESS_LOCATION = "":

# -- Variable:
# $SUCCESS_LOCATION
# -- Function:
# Defines the URL that users should be redirected to if the script works properly. If this is left blank, a default page will be returned to the user.

} # End of Configurable Options.

# Do Not Change Anything Below This Line. <-- #

chop $SAVE_DIRECTORY if ($SAVE_DIRECTORY =~ /\/$/);
use CGI qw(:standard);
$query = new CGI;

if ( !(e $SAVE_DIRECTORY) || !(-W $SAVE_DIRECTORY) || !(-d $SAVE_DIRECTORY) ) {
  print header;
  print "Content-type: text/html", "\n\n"; # i did this
  print <<_END_OF_HTML_CODE_

  _END_OF_HTML_CODE_

  }
The directory you specified:

```
\$SAVEDIRECTORY = "<B>$SAVEDIRECTORY</B;">;
```

is invalid. This problem is caused by one of the three following reasons:

- **The directory doesn't exist.** Make sure that this directory is a complete path name, not a URL or something similar. It should look similar to `<TT>/home/username/public_html/uploads</TT>`.

- **The directory isn't writable.** Make sure that this directory is writable by all users. At your UNIX command prompt, type `<TT>chmod 777 $SAVE_DIRECTORY</TT>`.

- **The directory you specified isn't really a directory.** Make sure that this is indeed a directory and not a file.
You attempted to upload a file that isn't properly formatted. The system administrator has decided that you can't upload files that begin with the word '<B>index</B>'. Please rename the file on your computer, and try uploading it again.

```php
# print header;
print "Content-type: text/html", "\n\n"; # ---- i did this
print "<HTML>
<HEAD>
<TITLE>Error: Filename Problem</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>Filename Problem</H1>
<P>You attempted to upload a file that isn't properly formatted. The file in question is <TT><B>$FILENAME_IN_QUESTION</B></TT> Please rename the file on your computer, and attempt to upload it again. Files may not have forward or backward slashes in their names. Also, they may not be prefixed with one (or more) periods.
<P>
</BODY>
</HTML>
```

```php
if (!open(OUTFILE, "$SAVE_DIRECTORY/$Filename")) {
    print "Content-type: text/plain\n\n";
    print "--------------------\n";
    print "Error:\n";
    print "--------------------\n";
    print "$SAVE_DIRECTORY/$Filename\n";
    print "--------------------\n";
    print "There was an error opening the Output File\n";
    print "for Writing.\n\n";
    print "Make sure that the directory:\n";
    print "$SAVE_DIRECTORY\n";
    print "has been chmodded with the permissions '777'.\n\n";
    print "Also, make sure that if your attempting\n";
    print "to overwrite an existing file, that the\n";
```
print "existing file is chmodded '666' or better.\n\n";
    print "The Error message below should help you
diagnose\n";
    print "the problem.\n\n";
    print "Error: $!\n";
    exit;
}

undef $BytesRead;
undef $Buffer;

while ($Bytes = read($FileHandle,$Buffer,1024)) {
    $BytesRead += $Bytes;
    print OUTFILE $Buffer;
}

push(@Files Written, "$SAVE_DIRECTORY/\$Filename");
$TOTAL_BYTES += $BytesRead;
%Confirmation{$File_Handle} = $BytesRead;

close($File_Handle);
close(OUTFILE);
chmod (0666, "$SAVE_DIRECTORY/\$Filename");
}

$FILES_UPLOADED = scalar(keys(%Confirmation));

if ($TOTAL_BYTES > $MAXIMUM UPLOAD && $MAXIMUM UPLOAD > 0) {
    foreach $File (@Files Written) {
        unlink $File;
    }

    #
    print header;
    print "Content-type: text/html", "\n\n"; # --- i did this
    print "<<_END_OF_HTML_CODE__;"

<!DOCTYPE HTML PUBLIC "-/W3C//DTD HTML 3.2//EN">
<html lang="EN">
<head>
    <TITLE>Error: Limit Reached</TITLE>
    <meta http-equiv="content-language" content="en">
    <meta http-equiv="content-style-type" content="text/css">
    <meta http-equiv="content-script-type" content="text/javascript">

    <style type="text/css">
    p { font-family : verdana, sans-serif;
        text-indent: 5%; margin-top: 30; margin-bottom: 0; margin-left: 5%;
        margin-right: 10; }
    p.normal { font-family : verdana, san-serif; }
    em { font-family : serif, times new roman ; font-size : medium ; font-weight : bold; }
    </style>
</head>

<body bgcolor="black" text="#bcbbaa" link="white" alink="yellow" vlink="gray">
Sorry, you have reached your upload limit set by the system administrator.

You attempted to upload $FILES_UPLOADED$ file(s),
totalling $TOTAL_BYTES$ bytes.
This exceeds the maximum limit of $MAXIMUM_UPLOAD$ bytes.

None of your file(s) were successfully saved.

Back Home
Upload Finished

You uploaded $FILES_UPLOADED file(s) totalling $TOTAL_BYTES total bytes.
Individual file information is listed below:

foreach $key (keys (%Confirmation)) {
    print "<li> $key - $Confirmation{$key} bytes\n";
}

print "\n";

exit;

# EOF
%%matlab file to access data from the RIPV VisualizeService

dataFile = 'data.dat';
inputFile = 'info.dat';
okFile = 'ok.dat';

while (1)
    % open the data file for reading
    % block until the file is created
    dfid = fopen(dataFile, 'r');
    disp('1st while')
    while(dfid == -1)
        dfid = fopen(dataFile, 'r');
        % wait for file to be written to
        pause(1);
    end
    disp('1st scan')
    % the data file will be two columns to begin with
    % if a more complex data file is required, change this line
    visData = fscanf(dfid, '%g %g', [2 inf]);
    % make the data a column vector
    visData = visData';
    fclose(dfid);

    % open the info file for reading
    % block until the file is created
    fid = fopen(inputFile, 'r');
    disp('2nd while')
    while(fid == -1)
        fid = fopen(inputFile, 'r');
        % wait for file to be written to
        pause(1);
    end
    % the input file will have the .jpg file name used for
    % storing the data output in
    outputFile = fscanf(fid,'%s');
    fclose(fid);

    plot(visData(:,1),visData(:,2));
    eval(['print -djpeg ' outputFile '.jpg'])

    % tell the java process that it that the vis is done
    oid = fopen(okFile, 'w');
    fprintf(oid,'1');
    fclose(oid);

    % delete the appropriate files
    % !del for PC
    % !rm for UNIX
    if strcmp(lower(deblank(computer)),'sgi'),
        DelCmd='!rm ';
    elseif strcmp(lower(deblank(computer)),'pcwin'),
        % We are running in W95 or NT
        DelCmd='!del ';
    elseif strcmp(lower(deblank(computer)),'lnx86'),
        % LINUX
        DelCmd='!rm ';
    else
        DelCmd='!del ';
    end

end
else
    disp(['computer ' controls not implemented.']);
end

%delete the files
eval([DelCmd dataFile])
eval([DelCmd inputFile])
end
SETUP.README for SUN station, Irix and Windows

**Table of Contents**

- Prerequisites
- Packages
- Directory Structure
- Locating Source Files
- Running the Compiler
- Locating Class Files
- JavaDoc - 1.2.2 and 1.3
- RMI
  - JARing class files
  - JARing comm.jar
  - UnJARing a File(s)
- Setting CLASSPATH
- Executing the Software

******************************

_Prerequisites_
This code should be run with at least jdk1.2 present on the underlying system

_Packages_
This is a brief explanation of where things are and how they got there. The classes have been sorted into packages. The current packages are listed in the 'ripv_api' directory.
They are:
Packages
mit.oe.deslab.rdamus.ripv
mit.oe.deslab.rdamus.ripv.assistants
mit.oe.deslab.rdamus.ripv.exceptions
mit.oe.deslab.rdamus.ripv.handlers
mit.oe.deslab.rdamus.ripv.managers
mit.oe.deslab.rdamus.ripv.messages
mit.oe.deslab.rdamus.ripv.panels
mit.oe.deslab.rdamus.ripv.services
mit.oe.deslab.rdamus.ripv.trees

_Directory Structure_
In order for the compiler to understand how to compile the classes, they must be located in a directory structure that is indicative of their package name. !You must pay attention to where you are storing files if you want the compiler to find them!
e.g. to find the RIPVServer now, you would import mit.oe.deslab.rdamus.ripv.RIPVServer; <-- for a source file
or
reference the RIPVServer like so at the command line:
\mit\oe\deslab\rdamus\ripv\RIPVServer.java <-- notice how the '\' in the command line takes the place of the '.' in the package name

_Locating Source Files_
Steps to locating the source files:
The source files are now located in the 'src' directory. This is relative to the current version directory:
e.g. current version 2.7.6
e:\thesis\ver2.7.6\src <--entry point for location of source files.
Thus, RIPVServer.java exists at:
e:\thesis\ver2.7.6\src\mit\oe\deslab\rdamus\ripv\RIPVServer.java

_Running the Compiler_
To run the compiler on the files, the only thing that *has* to be set is the PATH variable.
e:\thesis\ver2.7.6\>set PATH=c:\jdkl.3\bin <-- now you can use java, javac, rmic, etc...

Then to compile a particular file, you have to give the fully qualified directory name with respect to the current directory:
E:\thesis\ver2.7.6\>javac -verbose -sourcepath src -classpath c:\jdkl.3\lib\comm.jar src\mit\oe\deslab\rdamus\ripv\trees\*.java -d classes

The above example will compile all of the '.java' files in the mit.oe.deslab.rdamus.ripv.trees package.

the -verbose allows for easy debugging of where the compiler is missing (or getting) your files from
the -sourcepath is fed the top level of the directory where the packaged source files are located
note: because 'src' is given at the command line, it is with respect to the current directory.
This means there must be a directory called e:\thesis\ver2.7.6\src
the -classpath is pointing to the javax.comm.jar file which contains the appropriate files for doing the low level serial port handling.
the -d means to store the '.class' file in the directory called 'classes'
note: because '-d classes' is given at the command line, it is with respect to the current directory
This means there must be a directory called e:\thesis\ver2.7.6\classes

_Locating Class Files_
Steps to locating the class files
The class files are now located in the 'classes' directory. This is relative to the current version directory:
e.g. current version 2.7.6
e:\thesis\ver2.7.6\classes <--entry point for location of class files.
Thus, RIPVServer.class exists at:
e:\thesis\ver2.7.6\classes\mit\oe\deslab\rdamus\ripv\RIPVServer.class

_JavaDoc - 1.2.2 and 1.3_
In order to create documentation, you can either use the whole package or just the source files
The 'rmic' command is given at the command line with a fully qualified package name of a class as an argument. The tool will automatically generate the _Stub and _Skel and place them in the directory that the class file you are rmicompiling. It appears that wildcards like '*' cannot be used for rmicompiling all the classes in a particular package.

```
E:\thesis\ver2.7.6\classes>rmic mit.oe.deslab.rdamus.ripv.RIPVServer
BAD
E:\thesis\ver2.7.6\classes>rmic mit.oe.deslab.rdamus.ripv.services.*
```

_JARing class files_
The 'jar' tool is used at the top level of the directory hierarchy that contains the classes within it.

The command:
```
E:\thesis\ver2.7.6\classes>jar cf ripv.jar *
```

will cause a jar file to be named 'ripv.jar' within this jar file will be all of the .class files, in the correct directory structure as defined in the subdirectory structure of e:\thesis\ver2.7.6\classes.

Since the fully qualified name of a class file maps to the directory structure, then the jar file should mimic this in its containment hierarchy.

```
observe, the contents of ripv.jar:
-rw-rw-rw- 6035 11-Apr-2001 14:04:16
mit/oe/deslab/rdamus/ripv/services/DataVisServer.class
-rw-rw-rw- 7441 11-Apr-2001 14:04:16
mit/oe/deslab/rdamus/ripv/services/DatabaseServer.class
-rw-rw-rw- 5248 11-Apr-2001 14:04:16
mit/oe/deslab/rdamus/ripv/services/SerialPortServer.class
mit/oe/deslab/rdamus/ripv/services/DatabaseServerStub.class
mit/oe/deslab/rdamus/ripv/services/DatabaseServerSkel.class
mit/oe/deslab/rdamus/ripv/services/SerialPortServerStub.class
-rw-rw-rw- 3334 11-Apr-2001 15:33:24
mit/oe/deslab/rdamus/ripv/services/SerialPortServerSkel.class
```

113
_JARing comm.jar_

Copy the comm.jar file from c:\jdk1.3\lib to the current directory.
Proceed to extract the jar file into the current directory. The contents of the jar file will create a directory hierarchy.
Then, from the same directory that includes the subdirectories relating to the RIPV, type:
e:\thesis\ver2.7.6\classes>jar cf ripv.jar *

this will create a jar file, ripv.jar, that contains all of the necessary files for running the code.

_UnJARing a File(s)_

You can extract file(s) from a JAR file by using the ‘xf’ option:
C: \jar xf jar-file archived-file(s)


_Setting CLASSPATH_

The CLASSPATH variable can be set on a Unix machine running csh (know this because you have .cshrc file) like so:

setenv CLASSPATH <value>

or

set classpath=($classpath <value>)

_Executing the Software_

The software comes bundled in the ripv.jar JAR file. The directory structure of the jar file is described above. The way you execute the class files depends on what implementation of java you are running. I recommend downloading the Java 2 SDK, Standard Edition from Sun: http://java.sun.com/j2se/1.3. It is about 30MB in size and installs the latest version of the Java Virtual Machine. The Java Runtime Environment can be installed by itself, however, and only takes up ~5MB of disk space. It is located at http://java.sun.com/j2se/1.3/jre.

There are minor subtleties in how to execute the RIPV software based on your implementation, outlined below.

IMPORTANT:
In order to use the SerialPortServer’s serial port interfacing capabilities, the java.comm package must be installed on the system you are running the software on. This package is small (~270K) and is obtained from http://java.sun.com/products/javacomm.

Steps:
1. Install the implementation of Java (either JDK or plain JRE).
2. Install the appropriate files from the java.comm package in the appropriate directories. Read the platform-specific.html file that comes in the javadocs/ directory of the java.comm package for where to install the appropriate jar, dll and properties file. DO NOT forget to install the javax.comm.properties file or the SerialPortServer will NOT find any serial ports on your machine!!

3. Make sure that the PATH variable points to the location of where the ‘java’ command is:

if you are running the Java 2 SDK, version 1.3 then type the following command (Windows, see above for how to set environment variables in Unix):

C:\>set PATH=c:\jdk1.3\bin

for the JRE installation:

C:\>set PATH=c:\<JRE LOCATION>\jre\bin

4. Make sure that the CLASSPATH variable points to the location where the ripv.jar file is. This can be done in two ways:

set the CLASSPATH statically, using the directory where you have stored the ripv.jar file (in this example, in c:\ripv\classes):

C:\>set CLASSPATH=c:\ripv\classes\ripv.jar

or dynamically at run time using the -classpath option:

C:\>java -classpath c:\ripv\classes\ripv.jar <RIPV class name> <params>

Java will look for the files to run the software wherever it has been told by the classpath. By default, the classpath references the current directory if a classpath is not set as well as a default set of directories. For more information, see http://java.sun.com/products/jdk/1.2/docs/tooldocs/win32/classpath.html

4. Start the rmiregistry. This feature should be enabled on computers wishing to run the RIPVServer or RIPVServices. It is not needed for a machine connecting to the RIPVClient via a Web browser. If you do not start the rmiregistry, the RMI features of the RIPV system will not function.

5. Run the appropriate software; be sure to set the java.security.policy file (using the -D command line option) with the permit file that you have created. If you did not create a permit file, a default permit file can be extracted from the ripv.jar file. See the above section on UnJARing for how to do this. The fully qualified name of a class is used for execution.

In general, a java program is run by executing the .class file generated by the ‘javac’ compiler. Invoking the ‘java’ command at the command prompt, after setting the appropriate variables to tell the JVM where to look for the class files that are associated with this file, will cause the application to run.

RIPVServer
The RIPVServer is invoked with 1 argument parameter, the port on which to listen for requests from the RIPVClient. By convention, this port is 4444.

C:\> java -Djava.security.policy=permit mit.oe.deslab.rdamus.RIPVServer 4444

RIPVServices
The RIPVServices must know the location of the RIPVServer so they can register themselves. This argument will always be passed on the command line. For certain applications like the DatabaseServer, pertinent information about JDBC drivers is also passed to the application in addition to the RIPVServer’s location.

SerialPortServer
C:\> java -Djava.security.policy=permit mit.oe.deslab.rdamus.ripv.services.SerialPortServer //ripvserver.location

DataVisServer
C:\> java -Djava.security.policy=permit mit.oe.deslab.rdamus.ripv.services.DataVisServer //ripvserver.location

DatabaseServer
C:\> java -Djava.security.policy=permit mit.oe.deslab.rdamus.ripv.services.DatabaseServer //ripvserver.location [<driver> <url>]

The last two parameters are optional and can be neglected only if you wish to use the Cloudscape database that comes with the J2 Enterprise Edition. For more details about this Relational Database that uses SQL, see:

RIPVClient
Point your browser to the appropriate location of where the RIPV system is being hosted. For example:

http://deslab.mit.edu/ripv/RIPVClient.html
Appendix F - Class Hierarchy

Package Hierarchies:

mit.oe.deslab.rdamus.ripv, mit.oe.deslab.rdamus.ripv.assistants,
mit.oe.deslab.rdamus.ripv.exceptions, mit.oe.deslab.rdamus.ripv.handlers,
mit.oe.deslab.rdamus.ripv.managers, mit.oe.deslab.rdamus.ripv.messages,
mit.oe.deslab.rdamus.ripv.panels, mit.oe.deslab.rdamus.ripv.services,
mit.oe.deslab.rdamus.ripv.trees

Class Hierarchy

o class java.lang.Object

  o class java.awt.Component (implements java.awt.image.ImageObserver,
    java.awt.MenuContainer, java.io.Serializable)

    o class java.awt.Container

      o class javax.swing.JComponent (implements java.io.Serializable)

        o class javax.swing.JPanel (implements
          javax.accessibility.Accessible)

          o class mit.oe.deslab.rdamus.ripv.panels.DatabasePanel

          o class mit.oe.deslab.rdamus.ripv.trees.FileTree

          o class mit.oe.deslab.rdamus.ripv.panels.ImagePanel

          o class mit.oe.deslab.rdamus.ripv.trees.MailboxTree

          o class mit.oe.deslab.rdamus.ripv.trees.ServiceTree

        o class java.awt.Panel (implements javax.accessibility.Accessible)

          o class java.applet.Applet

            o class javax.swing.JApplet (implements
              javax.accessibility.Accessible,
              javax.swing.RootPaneContainer)

              o class mit.oe.deslab.rdamus.ripv.RIPVClient

            o class mit.oe.deslab.rdamus.ripv.CRCCCIT16

            o class mit.oe.deslab.rdamus.ripv.Footer (implements java.io.Serializable)

            o class mit.oe.deslab.rdamus.ripv.handlers.Handler (implements java.lang.Runnable)
- `ClientSocketHandler`
- `SerialPortHandler` (implements `javax.comm.SerialPortEventListener`)
- `SocketHandler`
- `Header` (implements `java.io.Serializable`)
- `idBadge` (implements `java.io.Serializable`)
- `MailBox` (implements `java.io.Serializable`)
- `MailBoxKey` (implements `java.io.Serializable`)
- `Message` (implements `java.io.Serializable`)
- `IncomingServerMessage`
- `IncomingServiceMessage`
- `OutgoingServerMessage`
- `OutgoingServiceMessage`
- `ObjectFIFO` (implements `java.io.Serializable`)
- `OutputStreamFlag`
- `PostOffice`
- `RemoteObject` (implements `java.rmi.Remote`, `java.io.Serializable`)
- `RemoteServer`
- `UnicastRemoteObject`
- `RIPVServer` (implements `mit.oe.deslab.rdamus.ripv.GenericRMInterface`, `java.langRunnable`)
- `RIPVService`
- `DatabaseServer` (implements `mit.oe.deslab.rdamus.ripv.GenericRMInterface`)
- `DataVisServer` (implements `mit.oe.deslab.rdamus.ripv.GenericRMInterface`)
- `SerialPortServer` (implements `mit.oe.deslab.rdamus.ripv.GenericRMInterface`)
- `ServiceReference` (implements `mit.oe.deslab.rdamus.ripv.GenericRMInterface`)
- `ServiceDescriptor` (implements `java.io.Serializable`)
- `SwingWorker`
- class java.lang.Thread (implements java.lang.Runnable)
  - class mit.oe.deslab.rdamus.ripv.assistants.InboxAssistantThread
  - class mit.oe.deslab.rdamus.ripv.managers.Manager
    - class mit.oe.deslab.rdamus.ripv.managers.GUIManager (implements mit.oe.deslab.rdamus.ripv.GUIInterface)
    - class mit.oe.deslab.rdamus.ripv.managers.TaskManager
  - class mit.oe.deslab.rdamus.ripv.assistants.OutboxAssistantThread
  - class java.lang.Throwable (implements java.io.Serializable)
    - class java.lang.Exception
      - class mit.oe.deslab.rdamus.ripv.exceptions.HandlerException

**Interface Hierarchy**
- interface mit.oe.deslab.rdamus.ripv.GUIInterface
- interface java.rmi.Remote
  - interface mit.oe.deslab.rdamus.ripv.GenericRMIInterface
References

ARION: http://dlforum.external.forth.gr:8080/

AUSI: http://www.ausi.org


Beaufort Sea 2: http://auvlab.mit.edu/history.html


A. G. Chappell, R. J. Komerska. An Environment for High-Level Multiple AUV Simulation and Communication. Autonomous Undersea Systems Institute, presented at


Freewave: http://www.freewave.com/Product Information/900_mhz.htm


Java Beans intro:

Java SUN: http://java.sun.com


Java Virtual Machine for Microsoft http://www.microsoft.com/java/default.htm


S. Lalis and A. Karipidis. An Open Market-Based Framework for Distributed Computing over the Internet. International Workshop on Grid Computing, in conjunction with the 7th International Conference on High Performance Computing (HiPC '2000), December 2000, Bangalore, India


NASA: http://ranier.oact.hq.nasa.gov/telerobotics_page/telerobotics.shtm


Onset: http://www.onsetcomputer.com/Products/Product_Pages/Tattletale_pages/TT8.html


Persistor: http://www.persistor.com

POSEIDON: http://czms.mit.edu/poseidon


SQL: http://www.sql.org


