Santa Clara University
DEPARTMENT of COMPUTER ENGINEERING

Date: June 9, 2004

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Carleton Cheng and Peter Salas

ENTITLED

RACE IV: Remote Accessible Control Environment

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER ENGINEERING

______________________	______________________
Dr. Neil Quinn THESIS ADVISOR	Dr. Christopher Kitts THESIS ADVISOR

____

Dr. Dan Lewis
DEPARTMENTCHAIR

RACE IV: Remote Accessible Control Environment

by

Carleton Cheng and Peter Salas

SENIOR DESIGN PROJECT REPORT

Submitted in partial fulfillment of the requirements
for the degree of
Bachelor of Science in Computer Engineering
School of Engineering
Santa Clara University

Santa Clara, California

June 9, 2004

Abstract

In space, there are hundreds of satellites orbiting the Earth. To talk to these satellites, communication ground stations need to be built in order to relay commands and data between the satellites and the human operators on the ground. Many large institutions, such as NASA, the European Space Agency, and the U.S. Air Force, have staff at full-scale remote communication stations. Furthermore, these stations are often distributed around the Earth in order to increase the amount of time that communications may take place given line-of-sight transmission constraints. Clearly, there is a need for "unmanned" ground stations that are controlled using human-in-the-loop remote operation coupled with some automation.

Santa Clara University is creating a series of communication ground stations under the framework of the Remote Accessible Communications Environment (RACE) system, which provides the software and tools to allow remote control of these stations. Many of these satellite-capable amateur radio communications stations are used typically by university satellite builders. This year's team re-designed the current ground station control system to handle better streaming information as well as to improve the remote control of the ground station. We accomplished successfully the following goals: a more versatile system to handle various interfaces, such as MATLAB and LabView; a more efficient method for remote control of the ground station; an improved latency of the system; and a successful installation and performance commanding the ground station from a remote location. The resulting system will advance the usage of this control network—which will assist distributed research and education for institutions, researchers, and students throughout the world.

Acknowledgements

To Dr. Neil Quinn and Dr. Chris Kitts for their help with the overall vision, planning, and design of the RACE project. We thank you for your invaluable guidance and assistance with our project.

A portion of this work was supported by the National Science Foundation under Grant No. EIA0079815 and EIA0082041; any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

— TABLE OF CONTENTS —

Page

Abstract …… i

Acknowledgements …… ii

Chapter 1 – Introduction …… 1

1.1 RACE – What is Race? …… 1

1.2 History …… 3

1.3 Problems …… 3

1.4 Project Goal …… 4

1.5 Contributions …… 4

Chapter 2 - Overall System Integration …… 6

2.1 System Overview …… 6

2.2 Team Structure …… 8

2.3 Design Process and Choices …… 8

Chapter 3 – Hardware …… 10

3.1 Current Hardware and Components Overview …… 10

3.2 Current Hardware Design …… 14

3.3 LabJack and Satellite Tracking …… 16

Chapter 4 - Software …… 17

4.1 Information Flow between Client and Server …… 17

4.2 Class Structure …… 18

4.3 Satellite Tracking Software (Predict) …… 18

4.4 RBNB DataTurbine …… 19

4.5 Transceiver …… 20

4.6 Serial Power Strip …… 21

4.7 Antenna Control Software …… 22

4.8 Packet Modem (TNC) …… 22

Chapter 5 - User Management …… 24

5.1 How to Install the System (Client Side)…… 24

5.2 How to Install the System (Server Side)……………………….……. 24
5.3 Computer Specification and Configuration (Client Side) ……………. 25
5.4 Computer Specification and Configuration (Client Side) ……………. 25
5.5 How to Use the System (Client Side) ………………………………... 26
5.6 How to Use the System (Server Side) ………………………………... 27
5.7 User Experience ………………………….. …………………….. 27
Chapter 6 – Experimentation and Testing ……………………………. 29
6.1 Testing Results ……………………………………………………….. 29
6.2 RACE Installation Metric ……………………………………………. 30
Chapter 7 – Societal Issues ……………………………………………….. 31
7.1 Ethical ………………………………………………………………... 31
7.2 Social ………………………………………………………………… 31
7.3 Political ……………………………………………………………… 31
7.4 Economic …………………………………………………………….. 32
7.5 Health and Safety ……………………………………………………. 32
7.6 Manufacturability ……………………………………………………. 32
7.7 Sustainability ………………………………………………………… 32
7.8 Environmental Impact ………………………………………………... 33
7.9 Usability …………………………………………………………… 33
7.10 Lifelong learning ……………………………………………………. 34
7.11 Compassion …………………………………………………………. 34
Chapter 8 – Conclusion …………………………………………………. 35
8.1 Summary …………………………………………………………… 35
8.2 Future Uses ………………………………………………………… 35
8.3 Future Contributions ………………………………………………….. 36
8.3 Lessons Learned ……………………………………………………… 37

Appendix …………………………………………………………………… 38

- Appendix A …………………………………………………………………… 40
  - Section A1: Client Side Class Structure ………………………….. 40
  - Section A2: Server Side Class Structure …………………………. 40
- Appendix B …………………………………………………………………… 41
  - Section B1: Client Input (Client-Side) ……………………………. 41
  - Section B2: DataTurbine (Client & Server Side) ………………… 44
- Appendix C …………………………………………………………………… 50
  - Section C1: Serial Power Strip Echo(Client-Side) ………………. 50
  - Section C2: Serial Power Strip Interpreter (Client-Side) ………… 54
  - Section C3: Serial Power Strip Server(Server-Side) …………….. 56
- Appendix D …………………………………………………………………… 61
  - Section D1: Predict Interpreter (Client-Side) ……………………. 61
  - Section D2: Predict Echo (Client-Side) ………………………….. 65
  - Section D3: Predict Server (Server-Side) ………………………… 67
  - Section D4: Predict (Server-Side) ………………………………... 70
  - Section D5: Predict Interpreter (Server-Side) ……………………. 73
- Appendix E …………………………………………………………………… 76
  - Section E1: Antenna Interpreter (Client-Side) …………………... 76
  - Section E2: Antenna Echo (Client-Side) ………………………… 79
  - Section E3: Antenna Server (Server-Side) ………………………. 81
  - Section E4: Antenna (Server-Side) ………………………………. 83
  - Section E5: Antenna Interpreter (Server-Side) …………………... 86
  - Section E6: AntennaAutoTracker (Server-Side) ………………… 89
- Appendix F …………………………………………………………………… 98
  - Section F1: Terminal Node Controller (Server-Side) ……………. 98
- Appendix G …………………………………………………………………… 104
  - Section G1: Transceiver (Client & Server Side) …………………. 104
  - Section G2: Transceiver Interpreter (Client-Side) ………………. 109
  - Section G3: Transceiver Echo (Client-Side) ……………………... 113
  - Section G4: Transceiver Server (Server-Side) …………………… 115

- Appendix H …………………………………………………………….. 123
    - Section H1: Server Side Directory Structure ……………………... 122
    - Section H2: Individual Batch Files ……………………………….. 122
    - Section H3: Main Batch File ……………………………………… 123
    - Section H4: Installation Instructions ……………………………… 124
    - Section H5: Using RACE Server Program ……………………….. 125
- Appendix I …………………………………………………………... 127
    - Section I1: Client Side Directory Structure ………………………. 127
    - Section I2: Individual Batch Files ………………………………... 127
    - Section I3: Main Batch File ………………………………………. 128
    - Section I4: Picture of RACE Client Program …………………….. 129
    - Section I5: Installation Instructions ………………………………. 130
    - Section I6: Using RACE Client Program ………………………… 131
    - Section I7: Matlab Configuration and User Manual ……………… 134
- Appendix J …………………………………………………………….. 136
    - Section J1: Use Case ……………………………………………… 136
    - Section J2: Use Case Descriptions ……………………………….. 137

— LIST OF FIGURES —

Figures

Figure 1.1: Ground-track of Sapphire (NO-45) with the Santa Clara ground station identified 2
Figure 2.1: RACE ground system architecture 6
Figure 3.1: ICOM 910 10
Figure 3.2: Kantronics 9612+ 11
Figure 3.3: Baytech RPC2 11
Figure 3.4: ICOM CI-V Level Converter 12
Figure 3.5: ICOM PS-125 Power Source 12
Figure 3.6: Yaesu G-5500 Antenna Controller 13
Figure 3.7: LabJack U-12 and LabJack PiggyBack 13
Figure 3.8: RACE Hardware Block Diagram 15
Figure 3.9 LabJack U12 16
Figure 4.1: Information Flow Diagram Between Client and Server Side Programs 17
Figure 4.2: Ground Station System Arcitecture with DataTurbine 19
Figure 4.3: MATLAB Interface 23
Figure 5.1: Hardware Configuration Diagram 26
Figure 6.1: Installation of RACE at NASA AMES Research Lab 30
Figure A-1: Client Side OM Diagram Structure 40
Figure A-2: Server Side OM Diagram Structure 40
Figure I-1: RACE Client Program 130
Figure J-1: Use Case Diagram 137

— LIST OF TABLES —

Tables

Table 1.1: University Missions Planning on Using RACE system ………………. 2

Table 6.1: Response Times of RACE System ……………………………………. 29

Chapter 1 - Introduction

1.1 RACE – What is RACE?

In space, there are hundreds of satellites orbiting the earth in order to provide us with benefits such as navigation signals, weather information, and phone/internet communications. These satellites are controlled by operators on the ground in order to manage the provision of these services and to remotely maintain the health of the spacecraft. To do this, commands and data must be relayed between the satellites and the human operators through the use of ground communication stations. These communication stations are often distributed around the Earth in order to increase the amount of time that communications may take place given line-of-sight transmission constraints. During communication sessions, human operators use these stations by logging remotely into them from one or more centralized operations facilities. For example, NASA[1], the European Space Agency[2], and the U.S. military all have a network of geographically distributed communication ground stations through which they communicate with large numbers of operational satellites.

Many universities build low-cost satellites and use ham radio communication stations in order to conduct command and data operations. Through the RACE[3] program, Santa Clara University (SCU) is developing a geographically distributed network of communication stations in order to operate many of the satellites built by SCU and its academic partners. Current RACE ground stations are in development and/or partially operational at SCU, in Hawaii, in Texas and in St. Louis. As depicted in Figure 1.1, it can be seen how this geographic distribution of communication stations dramatically increases the number of times each day that operators may communicate with university spacecraft in low Earth orbit.

---

[1] For more information about NASA, please see the following link: www.nasa.gov
[2] For more information about the ESA, please see the following link: www.esa.int
[3] RACE stands for Remote Accessible Control Environment

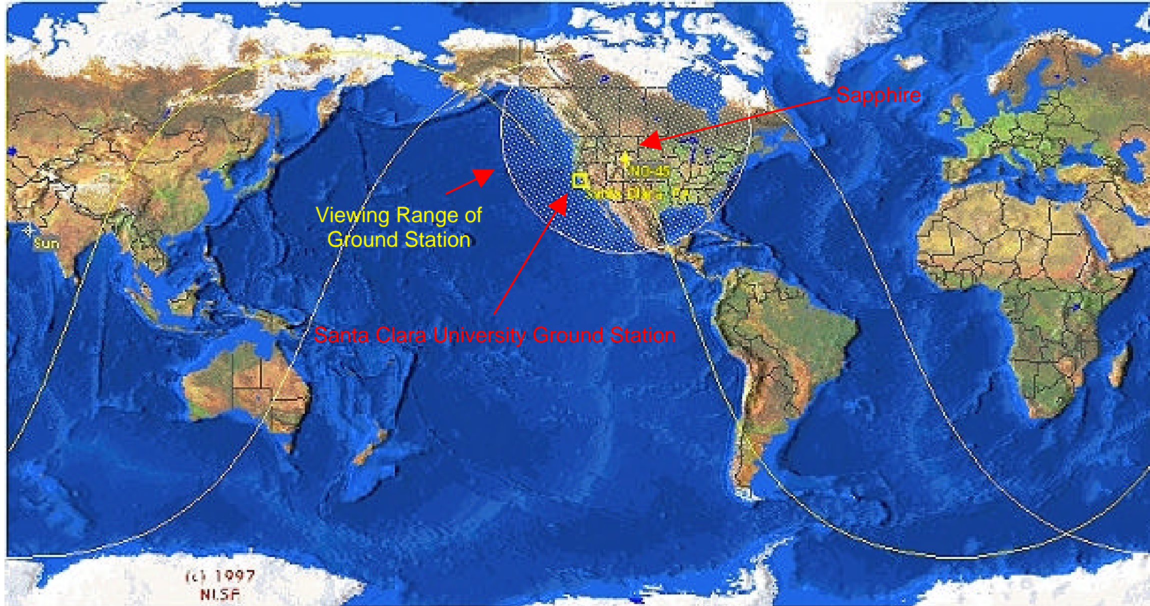

Figure 1.1: Ground-track of Sapphire (NO-45) with the Santa Clara ground station identified.

During this past academic year, the SCU RACE station has been used regularly to operate the Sapphire microsatellite. Table 1 summarizes the university missions planning on using the RACE system within the next 3 years.

Satellite Name	University	Mission	Status
Sapphire	Stanford	Photography, communications, component test, autonomy test	Operational on-orbit
Emerald	SCU, Stanford, MIT	2-satellite formation flying technology demonstrations	In development
FASTRAC	U.T. Austin, SCU	2-satellite formation flying technology demonstrations	In development
Akoya	Washington Univ. in Sty. Louis, SCU	Vision-based sensing and satellite inspection	In development
Genesat	NASA Ames, Stanford, SCU, Cal Poly	Astrobiology genetics experiment	In development, launch scheduled Nov 2005
CubeSats	Various Developers	Various experiments	Some in orbit, several launched yearly
FADSat	SCU	Hardware and automation testing	Operational in the lab

Table 1.1: University Missions Planning On Using RACE system

Over the past few years, RACE has evolved from a simple remote communications project into a web-controlled satellite communications environment. In its fourth year, we developed a better ground station system architecture.

1.2 History

Phase I of RACE began in 2000-2001 by the RACE I team—comprised of three electrical engineers. Their goal was to develop a Remote Accessible Communications Environment. The RACE I team successfully accomplished this by using a computer interface and HAM radio signals to communicate with a robotic device located in a remote location.

Phase II of RACE was headed by the RACE II team in 2001-2002—comprised of four computer engineers and two mechanical engineers. Building upon the accomplishments of the previous year, the RACE II team was able to expand the project into a system capable of communicating with satellites. In addition to this, team II developed a web-based satellite reservation system and built another control station located in Pearl City, Hawaii.

Phase III of RACE was lead by the RACE III team in 2002-2003—comprised of two computer engineers. This team added a web-based scheduler to the existing system, making it more of a web-controlled environment. This scheduler allows users to reserve a time-slot to communicate with a particular satellite. Team III also added an operator control feature to allow administrative rights to operators who manage the system.

1.3 Problems

For the past two years, the senior design teams demonstrated successfully that the web-based interface works. However, the interface needed considerable work to make it truly “operational.” This may be due to the fact that the online interface did not support streaming real-time information— which is vital for a system such as this. If a user wished to update the information in real-time, they must manually refresh the browser window. Time is extremely important because as the satellite orbits around the earth, it is only visible for a maximum of 15 minutes, and may not be in view again for another

several hours later. Information, such as if there is still a connection between the ground station and the satellite and the strength of this connection, is vital to the user.

The web interface also had many limitations. However, its biggest limitation was that users could not use their own program to send and receive data. The user was restricted to the web-based interface and typed commands in manually.

Another problem resided in the software architecture at the ground station computer. To control the ground station, the information was sent directly to the software, which handled that particular set of information. If this software were to change, the interface may need to be altered to accommodate for the software differences. Additionally, the satellite tracking software, Nova, posed problems. Because of Nova's properties, the user must either be physically at the ground station or use a remote desktop application, such as WinVNC, to command the satellite tracking software.

1.4 Project Goal

The main goal of this year's project was to administer a complete overhaul of the previous software architecture. Restructuring the system included: devising a better interface that supports streaming technology; making the ground station more versatile to support various interfaces and programs (i.e. MATLAB); replacing the satellite tracking software with one that could be controlled remotely; and building an overall system that is more efficient than the previous tracking software, Nova. All of these changes to the ground station were made without altering the system hardware because much of it was still intact and in excellent condition.

1.5 Contributions

Removing the current interface and replacing it with a more robust one, was the most likely approach. We incorporated streaming technology into the system to satisfy the need for a real-time interface. Furthermore, we introduced new technology without affecting or changing much of the existing infrastructure. Much time and research went into RACE's previous state and did not need to be changed dramatically.

Another feature we added to the ground station is the ability for a user to use his or her program to transfer data to and from a satellite. Incorporating streaming technology into the RACE ground station will enable users to "plug-into" the appropriate channel and stream data from their own choice of application on their computer. The user will not be limited to the confinements of a specific predefined interface.

Most of the work was done on the ground control station. We added another layer to the current software architecture to allow for better maintainability and upgradeability. By adding this additional layer of communication, we eliminated the problem of maintaining the interface, because of the various software used at the ground station. Therefore, if the software were to change, minimal amount of work will have to be done only to the additional layer itself.

Finally, we replaced the previous satellite-tracking software, Nova, with another one called Predict. Because Predict did not work with the satellite tracking hardware, we needed to replace the SASI Satellite tracker with LabJack, which is described in Chapter 4 of this thesis. All of these changes to the RACE system proved to make the ground station more remote accessible and more efficient.

Chapter 2 - Overall System Integration

2.1 System Overview

The RACE system allows the user to command a specified ground communication station located in a remote location. The goal of this system is to handle human-in-the-loop remote operation coupled with some automation of these ground stations. This schema helps to increase the amount of time that communications may take place given line-of-sight transmission constraints because many of these stations are often distributed around the Earth. The general system architecture is illustrated in Figure 2.1.

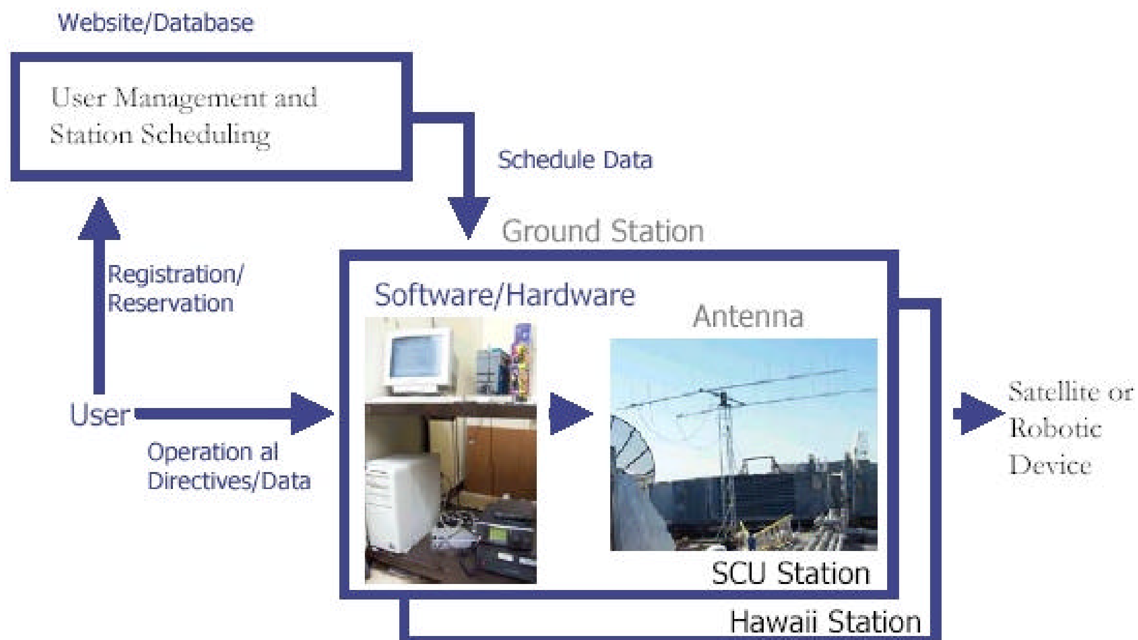

Figure 2.1: RACE ground system architecture [4]

Specifically, the user would be able to log in to the RACE system, and remotely command each satellite or robotic device to configure the system to their desired specifications. For example, if they wished to do so, users may send commands to set the appropriate frequency, in order to make contact with their satellite. The RACE system accomplishes contact with these satellites and robotic devices through the use of HAM

[4] Figure 2.1 was taken from the RACE 2002 Senior Thesis on pg. 5 of the following link: http://www.cse.scu.edu/send.cgi?srprojects/2003/COEN-2003-PROJECT-25.pdf

radio signals. In order to increase the efficiency and usability of the system, this year's RACE team administered a complete overhaul of the previous software architecture. Consequently, LabView was replaced by RBNB Data Turbine, which is described in Chapter 5, section 4.2 of this thesis. We chose RBNB Data Turbine because of its ability to handle streaming technology, its ability to support various programs, and its efficiency to handle data and networking between ground stations. The Java programming language was chosen as the primary language because RBNB Data Turbine is a Java-based application. Thus, using Java made it easier to integrate our component control software with RBNB Data Turbine.

To make the system truly remote accessible, we integrated LabJack and the Predict satellite tracking software into the system. To address the issue of the satellite tracking software aforementioned in Chapter 1, section 1.3, Predict was chosen because of Predict's ability to be controlled remotely. LabJack was needed because the SASI Satellite Tracking Hardware, used previously, was not compatible with Predict.

All of these components—RBNB Data Turbine, LabJack, our software and the previously established hardware—provides the user with a more remote accessible ground station to communicate with their satellite or robotic device. A typical user of RACE will first register with the RACE Reservation System[5] via the RACE website and the user's web browser. Upon administrative approval of their registration, the user can schedule an appointment to use one of the individual ground stations. Once they have scheduled a time to use the system, the user will return to the RACE control software or software of their choice. At this point the user may begin controlling the system by sending commands through the current command line interface to any of the hardware or to the satellite tracking software. After configuring the system to their specifications, the user can then use the RACE software or their own software to communicate with the selected satellites.

[5] The RACE Reservation System has not yet been integrated into our current system.

2.2 Team Structure

The team consisted of two computer engineers. The project was split up into different modules based upon the various equipment and technologies used. Because of the limited time frame for this project, a timeline of tasks was set-up to specify deadlines for completing each module. Each engineer picked a task from this timeline and worked until they were finished with their module. When it was complete, each engineer picked another task, and the timeline was modified. Communication throughout the project was easy because only two people were involved on the project. There was no specific group leader assigned because each took responsibility for their parts.

In lieu of this timeline, there were also weekly meetings with Dr. Neil Quinn and Dr. Christopher Kitts. These meetings proved to be important to address any problems we encountered and to insure that we met the necessary deadlines. These meetings helped clarify many issues concerning our project's design and implementation. Additionally, before beginning the project, we did not have any background in satellite communication. Drs. Quinn and Kitts helped us by providing insight into the different aspects of satellite technology, including the equipment used and how they are involved in satellite communication.

2.3 Design Process and Choices

The design process began in September of 2003. This year's RACE team met with project advisors Dr. Quinn and Dr. Kitts to discuss the concept the RACE project as well as to brief us on the history of RACE and the progress made by the previous teams. Additionally, we began laying out the basic requirements and goals of this project. Many ideas and designs were discussed until we decided on those aforementioned in chapter one. Once these requirements were laid out, we made some initial design specifications, concerning the overall system architecture, the programming languages and software to use, and any equipment needed.

After a formal design review in January, a few of these initial design specifications were altered and much of the implementation began. Over the course of the project, we found that some parts of the design needed to be altered slightly to

accommodate unforeseen problems with the equipment and software. Additionally, constant testing the software and hardware was necessary to ensure that the system worked well and with little error. However, much of the testing came after each part of the overall software architecture of the ground station system was complete.

Chapter 3 – Hardware

3.1 Current Hardware and Components Overview

Much of the equipment prior to this project was still intact and in excellent condition. Therefore, much of the current hardware did not need to be changed. However, because of the issues dealing with the remote control of NOVA, a new piece of hardware, called LabJack (described below) was added as a data-acquisition device to handle the antenna controller.

All electronic equipment is controlled through serial connections with the ground station computer. Using the new RACE software, every major device can be controlled, using the current command line interface.

- **Ground Station Computer**

Dell Optiplex GX110
Windows 2000 Professional
Pentium II – 128 MB RAM
4 Serial Ports, 1 Parallel Port

- **ICOM 910 Dual Band Transceiver**

The transceiver receives data from the packet creator, converts it to radio waves, and sends it out the antenna. It also receives radio waves and sends them to the packet creator for decoding. However, it must be set to the appropriate frequency before a connection can be achieved.

Figure 3.1: ICOM 910

- **Kantronics 9612 Packet Creator (Modem)**

A packet modem is needed to decode/encode satellite data transmissions. It also can receive commands from the user, using the software written by Daniel Shuet[6].

Figure 3.2: Kantronics 9612+

- **Baytech RPC2 Serial Port Controlled Power Strip[7]**

This power strip is used to remotely power on or off the various devices.

Figure 3.3: Baytech RPC2

---

[6] For more information on the code see Appendix F: Section F1
[7] The User Manual for this device can be found at the following link: http://www.baytech.net/downloads/manuals/U140E125-04_rpc.pdf

- **ICOM CI-V Level Converter**[8]

This device is used to communicate to the transceiver via the computer's serial port.

Figure 3.4: ICOM CI-V Level Converter

- **ICOM PS-125 Power Source**

A special power source is needed to power the transceiver.

Figure 3.5: ICOM PS-125 Power Source

[8] The notation CI-V is a standard naming convention given by ICOM America Inc.

- **Yaesu G-5500 Antenna Controller**

This component is the interface between the computer and the antenna rotator.

Figure 3.6: Yaesu G-5500 Antenna Controller

- **LabJack U12 and LabJack PiggyBack**[9]

Both of these pieces of equipment work in conjunction with Predict in order to control the antenna controller, which moves the antenna to the correct azimuth and elevation.

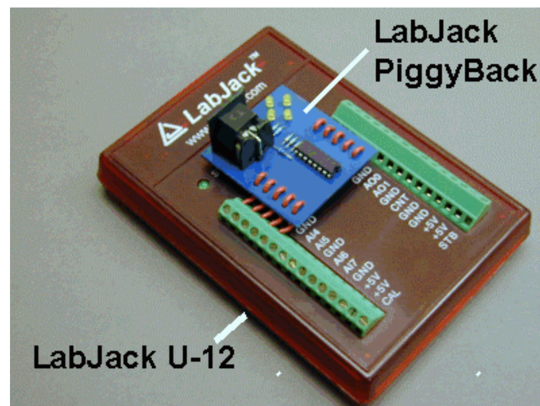

Figure 3.7: LabJack U-12 and LabJack PiggyBack

[9] For more information on LabJack and LabJack PiggyBack, refer to:
http://www.labjack.com/ and http://www.nlsa.com/labjack/labjack_piggyback.html

3.2 Current Hardware Design

Before initial design and implementation of this year's RACE system, we found that most of the hardware specified by the RACE II Senior Thesis[10] had still been intact. However, there were a few missing components and some areas of design that were not addressed. We needed to restructure the hardware block diagram not only to address these issues, but also to accommodate our own design. A new version of the hardware block diagram can be found in Figure 3.8.

[10] For more information on the RACE 2002 Senior Thesis hardware block diagram, please refer to Figure 4-8, pg. 33 at the following link: http://www.cse.scu.edu/send.cgi?srprojects/2003/COEN-2003-PROJECT-25.pdf

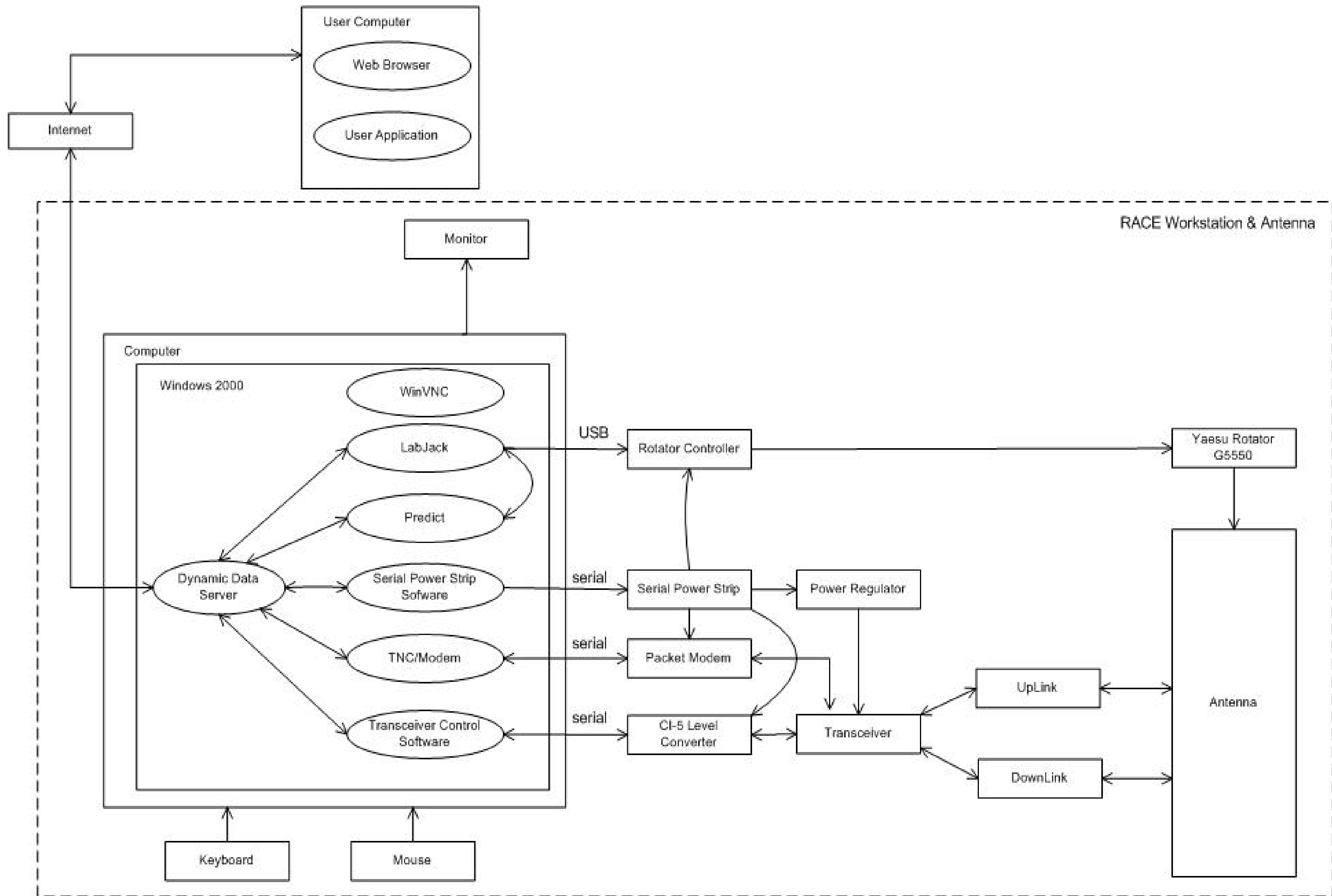

Figure 3.8: RACE Hardware Block Diagram

3.3 LabJack and Satellite Tracking

LabJack is a data-acquisition device, which is used to connect the PC to the antenna controller. In conjunction with the Satellite Tracking software, LabJack is used to direct the antenna toward the specified satellite in space.

Figure 3.9 LabJack U12

LabJack was chosen to replace the SASI Satellite Tracker[11]. LabJack and Predict—which is discussed in section 4.4— work together to provide a Satellite Tracking environment similar to Nova. The problem with the Nova software was that other software could not control it. Nova did not provide any means of controlling it remotely except using remote desktop software (i.e. WinVNC). Clearly, this is a security risk to the ground station, and therefore NOVA had to be replaced.

[11] Please see RACE 2002-2003 Senior Thesis, pg. 31 for further information and specifications SASI Satellite Tracker.

Chapter 4 - Software

4.1 Information Flow between Client and Server

Figure 4.1 is the information flow diagram between the Client and Server Side programs. It depicts how:

1) A client writes to a server
2) The ground station receives the message
3) The ground station writes the information to the devices
4) The ground station receives messages from the device
5) The ground station writes information out to receiveChannel
6) And the Client receives a response.

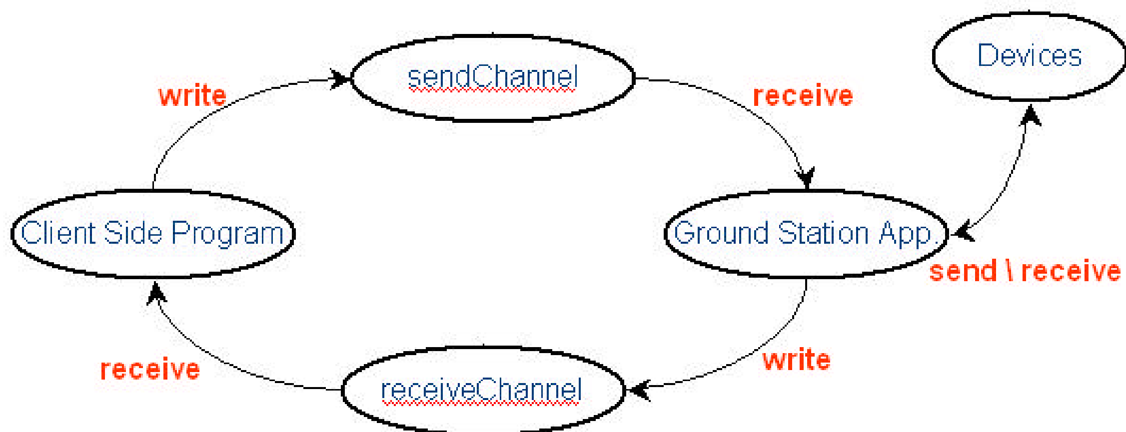

Figure 4.1: Information Flow Diagram Between Client and Server Side Programs

Most of our classes (except the TNC) follow this general information flow. Refer to this figure for the classes listed below.

4.2 Class Structure

Most of the classes follow a specific structure (Refer to *Appendix A*). All client side and ground station applications (server side) contain a DataTurbine class. This is needed to establish a connection with the DataTurbine server so that there is a communication link between the client and server side applications. Since each side will be accepting inputs, either from DataTurbine or from the user directly, both client and server side will have an interpreter that will validate commands and perform the appropriate operation.

Some of the programs may need to talk to a device connected to the serial port. Therefore, some of the classes will also need the Java Comm Port API[12] to be able to write and receive information from the Serial Port.

4.3 Satellite Tracking Software (Predict)

Predict[13] is the equivalent of the Nova program. The difference between them and also the primary reason why Predict was chosen is that Predict has a command-line based interface unlike Nova. This enables us, to call on predict from a Java program through a DOS-command prompt window and store the information from the window to a buffer.

Predict is a powerful program that serves our three important goals in a satellite tracking software:

- Remote updating of Keplerian elements used for calculating the position of a satellite
- Requesting the current position of a particular satellite
- Requesting the next pass over the ground station of a particular satellite.

Predict has numerous functionalities that can be performed from the dos-command prompt, however, only three main functions are supported right now.

- Predict –update predict.tle .... Updates Keplerian Elements
- Predict –poss <Satellite> .... Requests position of <Satellite>
- Predict –pass <Satellite> .... Requests next pass of <Satellite>

---

[12] The Java Com Port API can be found at http://java.sun.com/products/javacomm/
[13] Predict can be found and downloaded here: http://www.qsl.net/kd2bd/predict.html

4.4 Ring Buffered Network Bus (RBNB) DataTurbine[14]

RBNB (Ring Buffered Network Bus) DataTurbine is the heart of our system. It allows a user to talk to the devices through this one application. Previously, LabView served this purpose of integrating the ground station components into one application, but the problem with it was that it was not designed for real-time control of the ground station. The web-based interface was simply a mapped image of the LabView interface, therefore, in order for a user to get current information about the ground station and its devices, the user would have to refresh the browser.

RBNB is basically a dynamic data server that provides the framework for client/server applications. It serves as the foundation for data communication for multiple devices. It is also a java based program, thus we chose to implement most of our code in java to make the installation process simple and smooth.

DataTurbine talks to devices through a series of *channelmaps*. A *channelmap* is exactly what the name implies; a channel that both the client and server subscribe to and talk to each other on it. In our senior design, we have nine *channelmaps* defined:

- sendTransceiver & recieveTransceiver
- sendPredict & receivePredict
- sendPower & receivePower
- sendAntenna & receiveAntenna
- RXrawTNCString

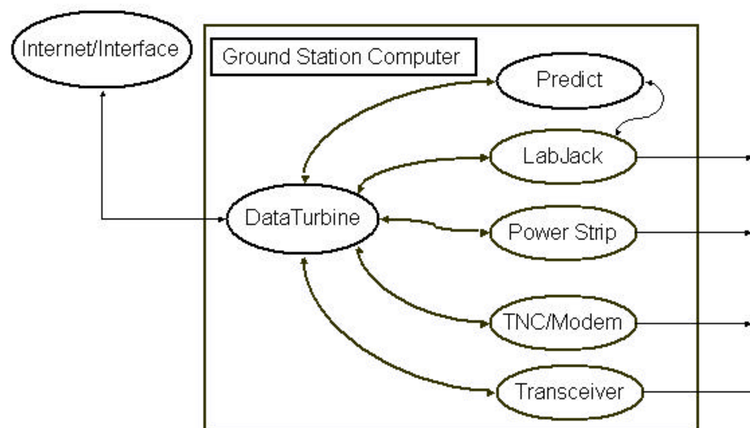

Figure 4.2: Ground Station System Architecture with DataTurbine

[14] More information on RBNB Data Turbine can be found at: http://rbnb.creare.com/

These *channelmaps* are used to forward information to and from the specific devices. The send channels are specifically used to send commands and information to the device, while the receive channels are information that is returned from the devices. It was designed this way so that both the sending and receiving of data can occur independently of the other. Creating two channels also ensures that there is no clash of information (sending at the same time from both client and server). Lastly, creating two channels enables the client side program to block and wait for information, rather than polling.

Also note that the *RXrawTNCString* was designed with only one *channelmap*. This was because there was an issue configuring Matlab with two channels. However, we have successfully coordinated the sending and receiving of information on the server side, so there will be no information lost (See *Appendix F-1* for source code).

4.5 Transceiver[15]

The transceiver is the device that sets the frequencies the packet modem will use to send and receive data. There are two frequency modes: uplink and downlink. The modem sends information out to the satellite through on the uplink frequency, and receives information from the satellite on the downlink frequency. Additionally, The CI-V level converter is also used—in conjunction with the transceiver—in order to interface with the computer. The CI-V level converter allows the programmer to be able to send and receive the necessary commands to control the transceiver.

When programming the transceiver one must understand the command format--which looks somewhat like this (which is used to set a frequency):

FE	FE	60	00	00	00	00154501	FD

Looking carefully, one will notice that:

1. Preamble (FIXED) : FE FE is the preamble which begins the command
2. Transceiver's default address: 60 (which can be changed on the transceiver hardware)

[15] For the complete user manual, refer to http://www.icomamerica.com/support/manuals/ic-910h.pdf

3. Controller's default address: 00 (which can be changed on the transceiver hardware)
4. Command number: 00 (specified by the command table[16])
5. Sub command number: 00 (specified by the command table)
6. BCD code data for frequency of memory number entry
   a. On closer inspection, on will notice that the frequency is reversed. In this case, the frequency was set to 145.150.0. Therefore, the BCD code data for frequency is formatted 00 15 45 01
7. End of message code (FIXED): FD is the value passed to the transceiver to tell it that the command is finished being sent.
8. NOTE: All commands are sent in hex format. In Java, on must use the function `parseInt` with base 16 in order to obtain the proper hex format sent through the serial port.

In general, programming the transceiver was a very difficult task because Java did not support the extended ASCII set[17].

4.6 Serial Power Strip

The Baytech RPC2 Serial Power Strip is a special power strip that can be controlled through the serial port. It was installed during the RACE 2001-2002 (RACE II) senior design. We connected RBNB DataTurbine to this device so that we can send commands to this device to turn on/off specific equipment. The devices that are connected to the Serial Power Strip are

- Antenna Controller
- Packet Modem (TNC)
- Transceiver
- CI-V Level Converter[18]

The code for the DataTurbine to Serial Power Strip can be found in *Appendix C*.

---

[16] For more information for programming the transceiver and CI-V level converter, refer to: http://www.plicht.de/ekki/civ/civtoc.html
[17] The extended ASCII table and information can be found at: http://www.zegelin.com/computers_files/ref/ACSII.htm
[18] CI-V Level Converter is needed to send commands from the PC to the Transceiver. Please reference RACE II Thesis: http://www.cse.scu.edu/send.cgi?srprojects/2002/COEN-2002-PROJECT-10.pdf

4.7 Antenna Control Software

Labjack[19] is a data acquisition device that is used in conjunction with Predict to provide an environment similar to Nova; we can predict when and where a satellite will come into view, and auto-track its position when it is in view. With the help of an Electrical Engineering graduate student, Dan Schuet, we successfully created a C executable file[20] called the *AntennaAutoTracker* for the sole purpose of controlling the *YAESU-G 5500*[21] antenna controller and integrated this program with *DataTurbine* to enable the transferring of information from server to client side.

Similarly to the *Predict* software, we can control the *AntennaAutoTracker* program through the DOS-command prompt and save any returned information to a buffer. We then write this buffer out to the *receiveAntenna* channel of *DataTurbine*, so that the user can view the information.

4.8 Packet Modem (TNC)

The Packet Modem (Also known as the TNC, Terminal Node Controller) is the hardware that is used to "talk" to a remote device. For the purpose of our project, it is used to communicate with Satellites. Because of this, the software had to be designed so that we can communicate with both the actual device at the ground station, and the remote device (i.e. a Satellite). It was decided that Dan Schuet, a graduate electrical engineering student, should handle the TNC communication, because of his knowledge of satellite communication. He designed it with only one channel to send and receive information that comes through the terminal node controller.

One of the RACE team's objectives was to allow multiple applications to interface with the system. One popular application that many universities use in their research is *Matlab*[22], thus it was important to make sure this application was supported. Due to lack of time, we were unsuccessful in designing *Matlab* to communicate with the

---

[19] Labjack and be found and purchased here: http://www.labjack.com/
[20] See *Appendix E-6* for the source code
[21] The antenna was installed by the RACE II team: http://www.cse.scu.edu/send.cgi?srprojects/2002/COEN-2002-PROJECT-10.pdf
[22] Matlab can be found at http://www.mathworks.com/

ground station with two channels. However, we have observed that Matlab handles well and there is no data loss with a single channel.

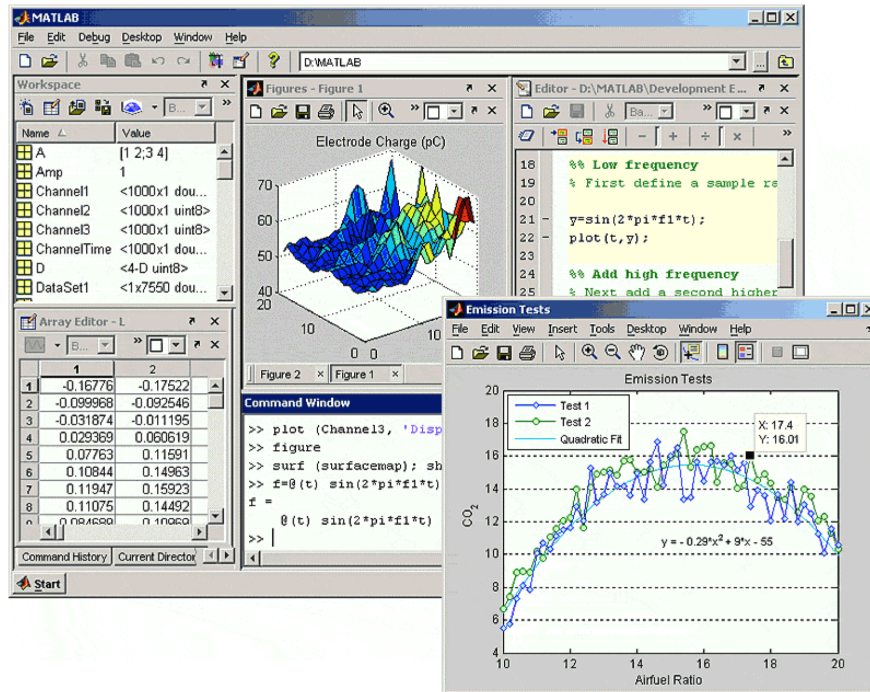

Figure 4.3: MATLAB Interface

Refer to *Appendix F* for the TNC source code.

Chapter 5 - User Management

5.1 How to Install the System (Client Side)

The software is pre-packaged in a zip file that contains a few batch files that will execute the necessary Java classes to control the ground station (Refer to *Appendix I* for information on the client side architecture and batch file scripts). There are a few configurations that a user will have to do prior invoking these class files. For complete instructions on installing and using the RACE Client Side System, please refer to *Appendix I-4 and I-5*.

Also, the user will have to install *Matlab* in order to send and receive information on the Packet Modem (TNC). Currently, the install process only contains direction for configuring *Matlab* to communicate with the Packet Modem (TNC). Future upgrades will hopefully allow other interfaces to interact with the Packet Modem (TNC). For complete instructions on configuring *Matlab* for the TNC, please refer to *Appendix I-6.*

5.2 How to Install the System (Server Side)

The server side software is prepackaged and designed to work with the following devices and hardware:

- *Labjack* data-acquisition and control device
- *ICOM CI-V Level Converter* connected to the *ICOM 910 Dual Band Transceiver*
- *Kantronics 9612 Packet Modem (TNC)*
- *Baytech RPC2 Serial Power Strip*

These devices must be installed and configured properly before the server side packaged software is installed and run. See below, *6.3.4 Computer Specification and Configuration (Server Side)*, for instructions on installing and configuring devices.

Once the hardware are installed to the right ports (see above) and configured properly, the prepackaged RACE server software can be installed and run. For complete instructions on installing and using the server software, please reference *Appendix H*.

5.3 Computer Specification and Configuration (Client Side)

The RACE client side system has been tested under the following computer specifications:

- Windows XP, 2000
- Internet Connection: 56K Modem (at least)
- 256 MB & 512 MB RAM
- Pentium II, III, IV Processor

Matlab needs to be installed on the client side machine in order to operate the TNC remotely. Because this program takes up a lot of memory, it is recommended that the computer have 512+ MB of RAM. Future upgrades of the client side software will eliminate this dependency.

Java Runtime Environment needs to be installed on the computer to be able to execute the Java programs. This can be downloaded for free from the java.sun.com website. It is recommended to install the JVM 1.4.x onto the computer.

5.4 Computer Specification and Configuration (Server Side)

The computer must have these initial specifications to install the devices to the computer:

- Windows OS
- Pentium II processor (at least)
  - Recommend Pentium III with 800+ MHz Processor
- 64 MB RAM (recommend 256+ MB or more)
- 3 Serial Ports
- 1 USB Port

Also, the software is designed to look for the device on a specific port:

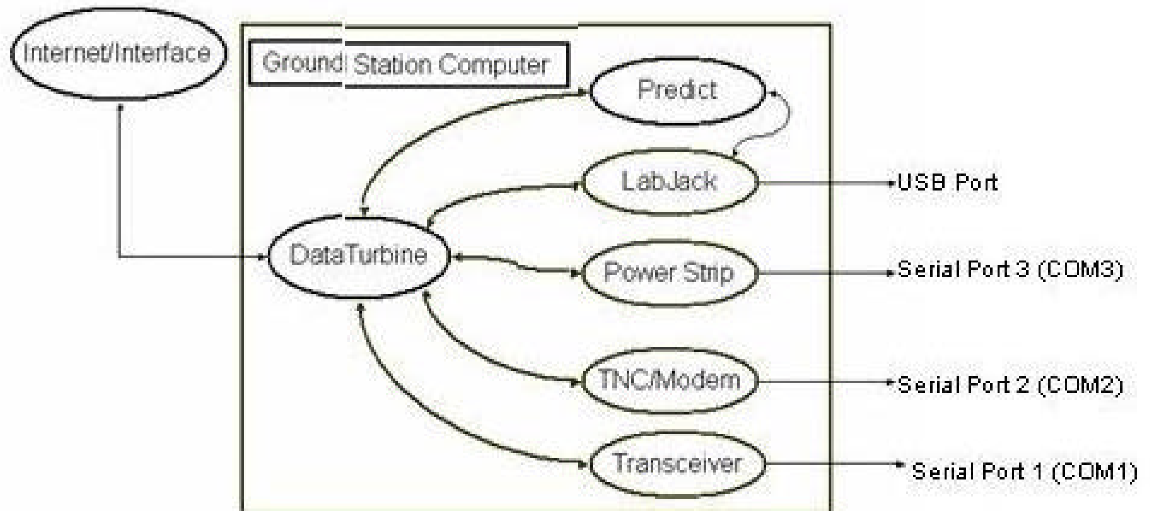

Figure 5.1: Hardware Configuration Diagram

Again, the prepackaged RACE server software is designed to work with the following devices[23]:

- *Labjack* data-acquisition and control device
- *ICOM CI-V Level Converter* connected to the *ICOM 910 Dual Band Transceiver*
- *Kantronics 9612 Packet Modem (TNC)*
- *Baytech RPC2 Serial Power Strip*

For more detailed instructions on how to install and configure the RACE server computer, see *Appendix H.*

5.5 How to Use the System (Client Side)

Consideration was taken about the user experience with the RACE program. Understanding that there is currently only a small community who are interested in communicating with satellites, and many of them prefer a command-prompt interface, we opted to make our interface a DOS-command prompt. A user has a better feel for how the RACE system works without any fancy buttons to distract or take away from the experience.

There are 5 command-prompt windows and *Matlab* that need to be opened to operate all functionalities of the RACE System:

- Power Strip Echo …… Outputs information from the Power Strip
- Transceiver Echo …… Outputs information from the Transceiver
- Predict Echo ………… Outputs information from the Satellite Tracking Software
- Antenna Echo ……….. Outputs information from *AntennaAutoTracker*[24] program
- Client Input ………… Accepts and Sends commands from the user to the Server
- *Matlab* Interface …….. Needed for control/access to Packet Modem (TNC)

To aid in invoking the 5 command-prompt windows, we made one master batch file that will start up all the command-prompt windows. For detailed instructions on using the RACE client program, see *Appendix I*.

5.6 How to Use the System (Server Side)

The system was designed to be very easy to install and run. Assuming that all components of the system were installed and configured correctly, an administrator in charge of maintaining the server would only need to invoke the main batch file located in the root folder of the packaged RACE server program. Detailed instructions on the operation and error handling are located in *Appendix H*.

5.7 User Experience

Unfortunately, we were unable to gather very much user experience data. We had gone over our deadline, and the complete system was not finished until mid May. This did not leave very much time for contacting potential users and asking them about their experience. However, professor Chris Kitts of the Santa Clara Engineering Department, and an undergraduate student, Dan Schuet, have tested the RACE system and their responses are positive.

According to Dr. Kitts, the performance of the RACE system is a big improvement over the previous system. The response times, he says, although relatively slow are negligible at this point and can probably be fixed by upgrading the ground station computer to a faster computer. He is also very pleased that the interface is mainly

---

[23] Many of the device specifications can be found on the RACE II Senior Thesis: http://www.cse.scu.edu/send.cgi?srprojects/2002/COEN-2002-PROJECT-10.pdf

[24] See *Appendix E-6*

DOS command-based. Overall, Dr. Kitts is very pleased with the improvements made on the system, and he will be using the system in the near future.

Dan Schuet is also pleased with the overall system. However, his only problem with the current Client Side program is that the TNC currently depends on *Matlab*. Although, he is very familiar with the *Matlab* program, he does not have a copy of it at home, therefore reducing his access. He hopes that future upgrades of the system will reduce this dependency, yet not eliminate the ability to use *Matlab*.

Chapter 6 – Experimentation and Testing

6.1 Testing Results

Unfortunately, this year's senior design team did not finish the complete system until late Spring 2004 quarter. This did not leave very much time for conducting a thorough system performance test nor gather user experience results.

We do have our day-to-day experience and generalized system response times. Below is a table with the average round-trip-times that we hand calculated for several typical ground station communication and control. These times reflect the responsiveness of the system from the initial sending of a command till a response is seen on the client side.

Device	**Average RTT (seconds)**
Packet Modem (TNC)	2.637
Serial Power Strip	2.578
Predict	1.856
Antenna	1.575
Transceiver	3.635

Table 6.1: Response Times of RACE System

NOTE: These times do not reflect the response time of sending and receiving a response from a Satellite. This time is dependent on strength of the link connection between the ground station and the satellite, and the natural propagation speed of radio waves through space.

We have also noted the overall system integrity. Within a one month test period of logging remotely into the ground station controls, there have had no system failures. In other words, we have not had to reboot the RACE server software. However, because the ground station runs on a Windows OS, it is recommended that the ground station

computer be formatted and rebooted every so often to maintain overall system performance.

The tests results gathered here are inconclusive, and we recommend further testing. There is a strong possibility that the system will be installed at the Hawaii Ground Station and other interested institutions. Hopefully, during the course of the summer, conclusive test results and figures will be collected.

6.2 RACE Installation Metric

Figure 6.1: Installation of RACE at NASA AMES Research Lab

We have also successfully installed the RACE system at the NASA Ames Laboratory in Moffet Field, CA. It was a large success because this was the first clean install of all the components of RACE. Approximate total time for the setup is about an hour. However, since we are familiar with the setup process, it may take more time.

Chapter 7 – Societal Issues

7.1 Ethical

RACE provides a means for underdeveloped areas cheap access to satellite technology. RACE is one solution in reducing the “Digital Divide,” by providing technology that previously was only available to the fortunate. Because it incorporates the internet, which serves as the framework for this network of groundstations, all educational communities can benefit from this project.

7.2 Social

Many social benefits result from the implementation of RACE. By giving students and faculty access to a satellite network such as RACE we are extending to them a very unique opportunity. Being the only non-government system of its kind, students are able to gain valuable experience in the field before becoming fully immersed in the industry. This unique project allows a greater number of people to influence this growing technology and help to improve it.

7.3 Political

Space technology, including ground control networks, is a technology that is regulated by the United States government. Although the impact has not been seen specifically by our group, the project development as a whole has been greatly affected. Originally, international ground stations were to be installed to enhance the capabilities of the RACE system and the amount of time that a satellite is in view of the network. These plans have been suspended due to lack of political approval.

7.4 Economic

RACE as a system allows for cost-effective and cheap operation of university satellites. Our contribution to this effort has significant economic ramifications in conducting student research, furthering hands-on education of satellites, and international student cooperation.

7.5 Health and Safety

RACE has no direct relation to health or safety in general. There are obvious safety precautions that should be followed when operating any equipment, including the equipment contained within the RACE system. Other than basic safety the RACE system does not have an effect on the health and safety of society as a whole or on individuals not associated with the use of the system. RACE could be used for research purposes, but the scope of the project does not focus on the various research topics that can be explored with the system.

7.6 Manufacturability

RACE is partially built. We have erected two stations, one at Santa Clara University in Santa Clara, California and the other in Pearl City, Hawaii. There are also two other schools, University of St. Louis and University of Austin, Texas, who are interested in the RACE program. These colleges have a partial system set up and are awaiting the software. If a school is interested, we also support the construction of a ground station on their site. Though the construction of each station is costly, there is much funding available for projects of this type. This project offers a great deal to the education community as a whole and provides exclusive opportunities for all parties involved.

7.7 Sustainability

RACE is now four years old and still in its infancy. There is much more work to be completed to obtain the full benefits of the RACE system. This project is sustainable for

many years to come as satellite networks become even more relied upon by our society. RACE allows for students to gain hands-on experience in an industry that is tightly watched by the government. RACE has been built to accommodate future growth. WE have implemented each part as a component to the whole which can be easily modified without reconstructing the entire system. This is an important aspect to the project as future years will want to focus on perfecting specific aspects of the RACE system without wanting to learn specifics about other components.

7.8 Environmental Impact

The RACE project as a whole has a large environmental impact, not in the classical sense of the earth, but on space. RACE is a network of satellites and ground stations. Each satellite that is put into orbit takes up some of the orbital space. As satellites go out of commission or break, they become space garbage, continually orbiting earth until eventually they burn up in the earth's atmosphere. Considerations must be made when a new satellite goes up to ensure that the satellite offers value to the university community. The second aspect of RACE that impacts the environment is the ground stations. Ground stations must be maintained. By building ground stations around the world, we must be sure that if the project were to end the ground stations be taken down or sold to be taken care of by another institution.

7.9 Usability

The entire focus of RACE IV is to implement a working and efficient communication link between the user and the ground station. RACE now has a much improved working user interface for ground station control and communication. In addition, the underlying technology used allows future design teams to implement a different interface on top of the system without much change to the existing code now. This year's design has successfully implemented a true real-time system enabling a user to log into the system and efficiently establish a link to their satellite.

7.10 Lifelong Learning

Neither of the group members had any knowledge of satellite functionality before starting this project. A learning curve was present for much of the first and second quarter. Not only did we have to learn about the individual components and their capabilities, we also had to learn about integration of the components with each other. Working on the RACE project has helped us to do independent thinking by problem solving on our feet at a fast pace. This project has taught us how to effectively and efficiently work within a team and how to accomplish a decent amount of work within a specified time. Problem solving, teamwork, and time management are not skills that can be obtained in the classic classroom setting. Through this project we have learned skills that will help us throughout our lives. We have learned how to teach our selves through research, allowing us to continue our learning process in the future.

7.11 Compassion

RACE does not not directly seek to relieve the suffering of others. Its main purpose is develop the framework for a network of groundstation for remote access control of robotic devices. However, since this is only a framework, there are many applications that can be applied to the project. Someone could essentially use our framework to provide a means of cheap communication to remote locations that need emergency help. RACE could be used for search and rescue operations to control rebotic devices in areas that are highly dangerous for humans.

Chapter 8 – Conclusion

8.1 Summary

This year's team enhanced the capabilities of the RACE system by administering a complete overhaul of the existing ground station software architecture. We accomplished this by removing LabView, Nova, and the SASI Satellite Tracker hardware and replacing them with RBNB DataTurbine, Predict, and the LabJack hardware, respectively. Additionally, because of our design, the RACE system is prepared to handle various user applications, making the system more versatile. By completing these tasks, we were able to improve the ground station's remote accessibility and efficiency.

Currently, through the RACE program, Santa Clara University is building a geographically distributed network of communication stations in order to operate many of the satellites built by SCU and its academic partners. Our software will be distributed and installed at various institutions such as NASA, the University of Hawaii, the University of Texas at Austin, and Georgia Tech. Once our software is distributed, installed, executed, and tested, the RACE project will begin the early phases of increasing the amount of time that communications may take place given line-of-sight transmission constraints. As a result, the RACE system will serve as the first layer of a global robotic control network that supports distributed research and education for students throughout the world.

8.2 Future Uses

The project has many uses, and can be extended to provide a multitude of others, such as allowing teachers to give their students the unique opportunity to experience controlling a satellite or remote vehicle. Additionally, this project may be used to control robotic devices for research in space, underwater, or other areas out of human reach. However, the current focus of this project is the remote control satellites—which have functions such as take pictures, collect data, and observe weather conditions of space.

One main use of the RACE system is to test experimentally new satellite operations techniques. Clearly, testing such techniques without a truly operational system poses much difficulty. Furthermore, systems operated by NASA ESA, and other

institutions are extremely complicated and the risky. Making mistakes on their system is too great and expensive to allow any good experimentation. Thus, RACE is a small-scale version of these large satellite operations networks comprised of components such as many groundstations, a central control facility, and numerous of users and satellites. Additionally, because this project is a small-scale model of a larger operation, we can "control" RACE to test out new operations techniques such as innovative fault diagnosis algorithms, new scheduling techniques, and other experiments.

8.3 Future Contributions

Future contributions to the RACE project may include things such as:

- *A Graphical User Interface (GUI) and incorporating the RACE Reservation System:* Currently, the system supports only a command line version. If the user does not have an application of their own, the RACE GUI will provide users with a more user-friendly interface. Additionally, the RACE Reservation System was not integrated into this year's project because of time constraints.

- *Web-Interface:* Seen as highly platform independent, a web-interface will allow users to utilize their web browsers to communicate with their satellite.

- *Integrating RACE with other satellite and robotic devices:* This will further expand the project into new areas of study as well as increasing the scope of the project.

- *Installation of More Ground Stations:* Although there is currently only one functional station, the Santa Clara ground station, more stations being built, which will increase the ground station coverage. Currently, educational institutions such as the University of Texas, Austin and Georgia Tech are constructing these ground stations. The Hawaii ground station located previously in Pearl City, Hawaii will soon be moved to the University of Hawaii, Manoa.

- *Satellites Controlled by RACE:* Currently, our focus is to test our system with the satellites. Much work must be done to extend the system to support the satellite missions listed in Table 1.1 as well as others.

8.4 Lessons Learned

The RACE project was truly an important experience. Learning about satellite communication, the equipment used in satellite communication, and the software involved in communication was very challenging. Although there were many aspects critical to this project, organization, problem-solving, and communication were the most important, in order to successfully accomplish our goals.

Before beginning this project, we were very inexperienced in satellite technology. Dr. Quinn and Dr. Kitts provided invaluable information and guidance to help us understand the inner workings of ground station and satellite communication. Once we had a handle on satellite technology, we sat down with the advisors to begin the design of the project. However, before we were ready to implement our project, we needed to obtain HAM radio licenses to be able to transmit/broadcast signals to an amateur satellite. Thus, there was much more learning to be done.

After becoming HAM licensees, we were able to begin implementation. Although we felt our design was solid, we still ran into some obstacles. We learned that the design process never ends. Our design needed to be revised every time there was either a better way to implement system or a slight flaw in the design. Additionally, we looked to the advisors for suggestions or assistance to troubleshoot problems with the equipment.

Finally, communications and compromise between teammates and advisors proved to be most vital to the project. In order to accomplish our project on time and solve problems in a quick and timely fashion, we needed to communicate constantly with each other and with our advisors. Additionally, because each of our styles of coding and implementation were different, compromises were necessary to keep the project on schedule. Communication also helped us to keep each of our implementation styles more uniform to ensure that the code was consistent and followed the design specification. Furthermore, the consistency of the code made maintaining the code and system much easier to handle.

— APPENDIX —

Appendix 38

- Appendix A 40
  - Section A1: Client Side Class Structure 40
  - Section A2: Server Side Class Structure 40
- Appendix B 41
  - Section B1: Client Input (Client-Side) 41
  - Section B2: DataTurbine (Client & Server Side) 44
- Appendix C 50
  - Section C1: Serial Power Strip Echo(Client-Side) 50
  - Section C2: Serial Power Strip Interpreter (Client-Side) 54
  - Section C3: Serial Power Strip Server(Server-Side) 56
- Appendix D 61
  - Section D1: Predict Interpreter (Client-Side) 61
  - Section D2: Predict Echo (Client-Side) 65
  - Section D3: Predict Server (Server-Side) 67
  - Section D4: Predict (Server-Side) 70
  - Section D5: Predict Interpreter (Server-Side) 73
- Appendix E 76
  - Section E1: Antenna Interpreter (Client-Side) 76
  - Section E2: Antenna Echo (Client-Side) 79
  - Section E3: Antenna Server (Server-Side) 81
  - Section E4: Antenna (Server-Side) 83
  - Section E5: Antenna Interpreter (Server-Side) 86
  - Section E6: AntennaAutoTracker (Server-Side) 89
- Appendix F 98
  - Section F1: Terminal Node Controller (Server-Side) 98
- Appendix G 104
  - Section G1: Transceiver (Client & Server Side) 104
  - Section G2: Transceiver Interpreter (Client-Side) 109

- Section G3: Transceiver Echo (Client-Side) …………………….. 113
- Section G4: Transceiver Server (Server-Side) …………………… 115

- Appendix H ……………………………………………………….. 123
  - Section H1: Server Side Directory Structure …………………... 122
  - Section H2: Individual Batch Files …………………………….. 122
  - Section H3: Main Batch File ……………………………………. 123
  - Section H4: Installation Instructions …………………………… 124
  - Section H5: Using RACE Server Program ……………………….. 125
- Appendix I ………………………………………………………... 127
  - Section I1: Client Side Directory Structure ………………………. 127
  - Section I2: Individual Batch Files ……………………………... 127
  - Section I3: Main Batch File ……………………………………. 128
  - Section I4: Picture of RACE Client Program …………………….. 129
  - Section I5: Installation Instructions ……………………………. 130
  - Section I6: Using RACE Client Program ………………………… 131
  - Section I7: Matlab Configuration and User Manual ……………… 134
- Appendix J ………………………………………………………….. 136
  - Section J1: Use Case ……………………………………………. 136
  - Section J2: Use Case Descriptions ……………………………….. 137

Appendix A

Section A1: Client Side Class Structure

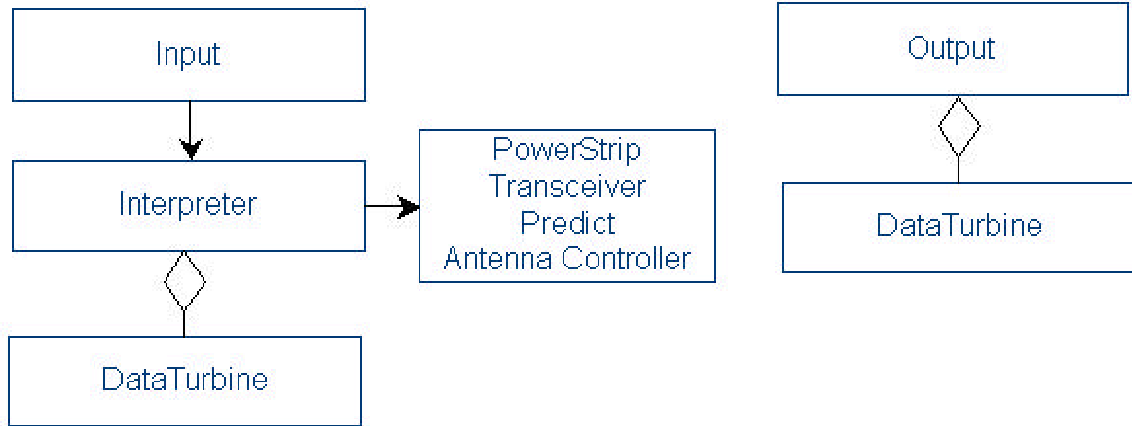

Figure A-1: Client Side OM Diagram Structure

Section A2: Server Side Class Structure

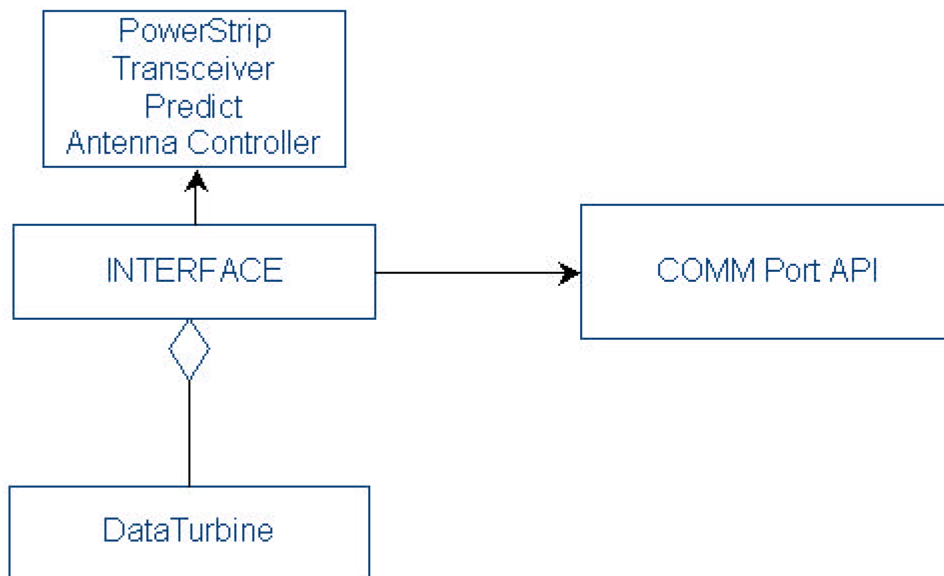

Figure A-2: Server Side OM Diagram Structure

Appendix B

Section B1: Client Input (Client Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto:PSalas@scu.edu"> (PSalas@scu.edu) </A>
 * <P>
 * @version
 * Created: 4/13/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class integrates all
 **/

import java.io.*;
import java.util.*;
import java.lang.*;
import com.rbnb.sapi.*;

public class ClientInput
{
      static String messageString = null;

      public static void main(String [] args)
      {
            String host;
      //Variable used to specify where DT Host is
            PredictClientInterpreter predictInterpreter;
      //Interpreter for Predict
            PowerStripClientInterpreter powerInterpreter;
      //Interpreter for Power Strip
            TransceiverInterpreter xcrInterpreter;
      //Interpreter for Transceiver
            AntennaClientInterpreter antennaInterpreter;
      //Interpreter for Labjack (Antenna Controller)

            System.out.println("RACE: Command Input Screen\n");
            if(args.length>0)
            {
                  host=args[0];
                        //user must enter in the IP of DT Server
                  System.out.println("Connecting to " + host + "...");
            }
            else
            {
                  System.out.println("ERROR: No RBNB host defined");
                  return;
            }

            //establish a connection with all the equipment located
```

```
            //at the Ground Station
            predictInterpreter = new PredictClientInterpreter(host);
            powerInterpreter = new PowerStripClientInterpreter(host);
            xcrInterpreter = new TransceiverInterpreter(host);
            antennaInterpreter = new AntennaClientInterpreter(host);

            while (true)
            {
                  /******     Command Input     ******/
                  InputStreamReader isr = new InputStreamReader (
System.in );
                  BufferedReader br = new BufferedReader ( isr );
                  messageString = null;
                  try
                  {
                        System.out.print("Please Enter Command: ");
                        if ( (messageString = br.readLine ()) != null )
                        {
                                    //validate command for Predict
                              if
(predictInterpreter.validateCommand(messageString)) {

      predictInterpreter.sendCommand(messageString,1);
                              }
                                    //validate command for Serial Power
Strip
                              else if
(powerInterpreter.validateCommand(messageString)) {

      powerInterpreter.sendCommand(messageString,1);
                              }
                                    //validate command for Transceiver
                              else if
(xcrInterpreter.validateCommand(messageString)) {

      xcrInterpreter.sendCommand(messageString,1);
                              }
                                    //validate command for Labjack
(Antenna Controller)
                              else if
(antennaInterpreter.validateCommand(messageString)) {

      antennaInterpreter.sendCommand(messageString,1);
                              }

                              else {
//error message if the user does not enter in a valide command for
//devices
                                    System.out.println("\nERROR: If you
need help, try typing in\n\n   predict|xcr|power|antenna -help\n");
                              }

                        }
                  }
                  catch(Exception e){ System.out.println("This did not
work\n"); }
            }
```

```
        }
}
```

Section B2: DataTurbine (Client & Server Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto:Psalas@scu.edu"> (PSalas@scu.edu) </A>
 * <P>
 * @version
 * Created: 2/16/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class integrates all the basic and advanced
 * functionality of Creare RBNB DataTurbine.  It can be used to create
 * a Sink, Source, or Plugin connection with a DT server.  It also
 * contains the four ChannelMaps that are used for communicating with
 * the DT server: sending, receiving, accessing, and the PlugIn
 * ChannelMap.  For more information on DataTurbine and it's
 * functionality, please refer to the RBNB website:
 * http://rbnb.creare.com
 * <P>
 * <B>NOTE:</B> This version of DataTurbine.java does not have any
 * security (i.e. username and password) written into the channels.
 * Basically, anyone can log onto the channel at any time.
 **/

import java.io.*;
import java.util.*;
import java.lang.*;
import com.rbnb.sapi.*;

public class DataTurbine
{

/*********************************************************************/
/***************       Constructor       ****************************/
/*********************************************************************/

/**
* Description: initializes the variables for DataTurbine
*     communication.<P>
* Precondition: none<P>
* Postcondition: all variables are initialized to its default value.
**/
      public DataTurbine()
      {
            plugin = new PlugIn();
            source = new Source();
            sink = new Sink();
            sMap = new ChannelMap();
            aMap = new ChannelMap();
            rMap = new ChannelMap();
            sNum = -1;
            aNum = -1;
            rNum = -1;
      }
```

```
/*******************************************************************/

/**************          Accessors & Mutators          ****************/
/*******************************************************************/

/**
* Description: opens a plugin connection given the address of the rbnb
*     server, and the name of the plugin the user wants to open.<P>
* Precondition: user must provide the address of the rbnbserver host,
*     and the name of the plugin that the user wants to establish.<P>
* Postcondition: function will attempt to establish a connection. else,
*     the function will print out the stack trace error.
**/
      public void openPluginConnection(String rbnbServer, String
pluginName)
      {
            try
            {
                  plugin.OpenRBNBConnection(rbnbServer, pluginName);
            }
            catch(SAPIException se) { se.printStackTrace(); }
      }

/**
* Description: open a source connection given the address of the rbnb
*     server, and the name of the source the user wants to open.<P>
* Precondition: user must provide the address of the rbnbserver host,
*     and the name of the source that the user wants to establish.<P>
* Postcondition: function will attempt to establish a connection, else,
*     the function will print out the stack trace error.
**/
      public void openSourceConnection(String rbnbServer, String
sourceName)
      {
            try
            {
                  source.OpenRBNBConnection(rbnbServer, sourceName);
            }
            catch(SAPIException se) { se.printStackTrace(); }
      }

/**
* Description: open a sink connection given the address of the rbnb
*     server, and the name of the sink the user wants to open. <P>
* Precondition: user must provide the address of the rbnbserver host,
*     and the name of the sink that the user wants to establish. <P>
* Postcondition: function will attempt to establish a connection, else,
*     the function will print out the stack trace error.
**/
      public void openSinkConnection(String rbnbServer, String
sinkName)
      {
            try
            {
                  sink.OpenRBNBConnection(rbnbServer, sinkName);
            }
```

```
		catch(SAPIException se) { se.printStackTrace(); }
	}

/**
* Description: adds a sendChannel<P>
* Precondition: channelName is of type String<P>
* Postcondition: attempts to add a channel.  else, the function will
*	print out the stack trace error.
**/
	public void addChannelSend(String channelName)
	{
		try
		{
			sNum = sMap.Add(channelName);
		}
		catch(SAPIException se) { se.printStackTrace(); }
	}

/**
* Description: adds a receiveChannel<P>
* Precondition: channelName is of type String<P>
* Postcondition: attempts to add a channel.  else, the function will
*	print out the stack trace error.
**/
	public void addChannelReceive(String channelName)
	{
		try
		{
			rNum = rMap.Add(channelName);
		}
		catch(SAPIException se) { se.printStackTrace(); }
	}

/**
* Description: adds a accessChannel<P>
* Precondition: channelName is of type String<P>
* Postcondition: attempts to add a channel.  Else, the function will
*	print out the stack trace error.
**/
	public void addChannelAccess(String channelName)
	{
		try
		{
			aNum = aMap.Add(channelName);
		}
		catch(SAPIException se) { se.printStackTrace(); }
	}

/**
* Description: gives the index of the sendChannel<P>
* Precondition: the user has added a sendChannel<P>
* Postcondition: the index of the sendChannel is returned.
**/
	public int getIndexSend()
	{
		return sNum;
	}
```

```
/**
* Description: gives the index of the receiveChannel<P>
* Precondition: the user has added a receiveChannel<P>
*     Postcondition: the index of the receiveChannel is returned.
**/
      public int getIndexReceive()
      {
            return rNum;
      }

/**
* Description: gives the index of the accessChannel<P>
* Precondition: the user has added a accessChannel<P>
* Postcondition: the index of the accessChannel is returned.
**/
      public int getIndexAccess()
      {
            return aNum;
      }

/**
* Description: registers the receiveChannel to a plugin<P>
* Precondition: a plugin connection must be opened, and a
*     receiveChannel has to have been added already.<P>
* Postcondition: function attempts to register receiveChannel to
plugin.
*     Else, the function prints out the stack trace error.
**/
      public boolean registerReceiveChannel()
      {
            try
            {
                  plugin.Register(rMap);
                  return true;
            }
            catch(SAPIException se) { se.printStackTrace(); return
false;}
      }

/**
* Description: sends a message to DataTurbine<P>
* Precondition: a sendChannel needs to have already been added.  A sink
*     connection must be opened. The value wait is the number of
*     miliseconds DT should wait for a response.  If the value is
*     negative (i.e. -1), the DT will wait for a response infinite
*     amount of time (blocking wait).<P>
* Postcondition: function attempts to send message. Else, the function
*     prints out the stack trace error.
**/
      public void sendMessage(String message, int wait)
      {
            try
            {
                  sMap.PutDataAsString(sNum,message);
                  sink.Request(sMap,0,0,"newest");
                        //parameters are irrelevent
```

```
                ChannelMap cm2=sink.Fetch(wait);
            }
            catch(SAPIException se) { se.printStackTrace(); }
      }

/**
* Description: checks to see if there is a message that was written
*     into the plugin. if there is, it pulls the message out and
*     returns it back to the user.
* Precondition: must open a plugin connection.  The message needs to be
*     of type String.  Also, the receiveChannel has to have been
*     registered to the plugin.  The value wait is the number of
*     miliseconds DT should wait for a response.  If the value is
*     negative (i.e. -1), the DT will wait for a response infinite
*     amount of time (blocking wait).
* Postcondition: the message is returned to the user if there is a
*     message. Else, the message "<not ready>" will be returned if
*     there is no new data to be taken from the DT server.
**/
      public String receiveMessage(int wait)
      {
            try
            {
                  String message = null;

                  PlugInChannelMap picm=plugin.Fetch(wait);
                  picm.PutTime((double)System.currentTimeMillis(),0);
                  //extract message, silently ignore nontext messages

                  if(picm.NumberOfChannels()>0)
                  {
                        if (picm.GetType(0)==ChannelMap.TYPE_STRING)
                        {
                              message = picm.GetDataAsString(0)[0];

                              String response="Message Received.";
                              picm.PutMime(0,"text/plain");
                              picm.PutDataAsString(0,response);
                              plugin.Flush(picm);
                        }
                  }
                  plugin.Flush(picm);

                  return message;
            }
            catch(SAPIException se) { se.printStackTrace(); return
"<not ready>";}
      }

      private PlugIn plugin = null;        // Dynamic Source
      private Source source;               // Source to write to DT
      private Sink sink;             // Sink to process requests to DT
      private ChannelMap sMap, aMap, rMap;
            // ChannelMaps for sending, receiving, and accessing
      private PlugInChannelMap picm;
            // An extended version of a ChannelMap for a PlugIn
      private int sNum, aNum, rNum;
```

```
        // Index values of the ChannelMaps
}
```

Appendix C

Section C1: Serial Power Strip Interpreter (Client-Side)

```
/**
 * @Author: Peter Salas
 *    <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 2/20/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class is used for validating user inputs, and
 * interpreting
 * inputs to the actual commands that the Ground Station accepts for
 * remote access. Specifically, this class validates commands for the
 * Power Strip. To establish a connection with the Ground Station
 * located at Santa Clara University Engineering Department. For more
 * information on DataTurbine and its functionality, please refer to
 * the RBNB website: http://rbnb.creare.com
 **/

import java.io.*;
import java.util.*;

public class PowerStripClientInterpreter {

/*********************************************************************/
/*******************         Constructor       *********************/
/*********************************************************************/

/**
* Description: initializes the mapped variables for the control of the
*     Labjack (Antenna Controler). This constructor also establishes a
*     connection to the Ground Station using DT, so that it can send
*     commands to the device.<P>
* Precondition: The IP of the DT Server must be given to establish a
*     connection<P>
* Postcondition: all variables are initialized to its default value. If
*     a connection was made successfully to the DT Server, this
*     constructor will return with no errors.  If there is an error
*     connecting, then the constructor will return with an RBNB
*     DataTurbine error.
**/
   public PowerStripClientInterpreter(String host) {
      commandArray[0] = "power -on lvconvert";
      commandArray[1] = "power -off lvconvert";
      commandArray[2] = "power -on xcr";
      commandArray[3] = "power -off xcr";
      commandArray[4] = "power -on modem";
```

```
      commandArray[5] = "power -off modem";
      commandArray[6] = "power -on antenna";
      commandArray[7] = "power -off antenna";
      commandArray[8] = "power -help";
      commandArray[9] = "power -on";
      commandArray[10]= "power -off";

      dt.openSinkConnection(host,"PowerStripClientSink");
      dt.addChannelSend("/sattest/sendPowerStrip/text");
      dt2.openSinkConnection(host,"PowerStripClientSink2");
      dt2.addChannelSend("/sattest/receivePowerStrip/text");
   }

/*********************************************************************/
/*************          Accessors & Mutators          ***************/
/*********************************************************************/

/**
* Description: This function validates a user command.<P>
* Precondition: The user input must be sent of type String.<P>
* Postcondition: If the command is a valide command, this function
*     return true. Else, it returns false
**/
   public boolean validateCommand(String command) {
      return (getCommandIndex(command) != -1);
   }

/**
* Description: This sends the command to the Ground Station<P>
* Precondition: The command from the user must be sent of type
*     String<P>
* Postcondition:  If the command is a valid command, then the command
*     is sent and function returns true. Else, the function returns
*     false.
**/
   public boolean sendCommand(String command, int wait) {
      int index = getCommandIndex(command);

      switch (index) {
            case 0: dt.sendMessage("ON 1",wait);
                  return true;

            case 1: dt.sendMessage("OFF 1",wait);
                  return true;

            case 2: dt.sendMessage("ON 3",wait);
                  return true;

            case 3: dt.sendMessage("OFF 3",wait);
                  return true;

            case 4: dt.sendMessage("ON 6",wait);
                  return true;

            case 5: dt.sendMessage("OFF 6",wait);
                  return true;
```

```
            case 6: dt.sendMessage("ON 4",wait);
                  return true;

            case 7: dt.sendMessage("OFF 4",wait);
                  return true;

            case 8:
//sends the command to the receivePowerStrip channel instead of
//sendPowerStrip channel
                      dt2.sendMessage(getHelp(),wait);
                  return true;

            case 9: dt.sendMessage("ON",wait);
                  return true;

            case 10: dt.sendMessage("OFF",wait);
                  return true;
            default: return false;
      }
   }

/**
* Description: This goes through the mapped commands of the Power
Strip,
*     and returns the index of that command.<P>
* Precondition: The user command must be sent of type String.<P>
* Postcondition: If the command is valid, then this function returns
*     the index of the command. If the command is not valid, this
*     function returns -1.
**/
   private int getCommandIndex(String command) {
      for(int i=0; i<numCommand; i++) {
            if (command.toLowerCase().startsWith(commandArray[i]))
                  return i;
      }

      return -1;
   }

/**
* Description: This is the help menu for the Power Strip<P>
* Precondition: No precondition<P>
* Postcondition: returns the help menu.
**/
   public static String getHelp() {
      String help1 = "-------------------------\nHelp File for Power
Strip\n-------------------------\n\n";
      String help2 = "commands:\n   -on\n\n   -off\n\n   -on
lvconvert\n\n   -off lvconvert\n\n";
      String help3 = "   -on xcr\n\n   -off xcr\n\n   -on modem\n\n   -
off modem\n\n";
      String help4 = "   -on antenna\n\n   -off antenna\n\n   -help";

      String help = help1 + help2 + help3 + help4;

      return help;
   }
```

```
/**
* Description: This counts the number of arguments in a message of type
*     String<P>
* Precondition: A message of type String must be sent<P>
* Postcondition: the number of words in a message is returned<P>
* NOTE: Words are deliminated by spaces.
**/
   private static int countArguments(String message) {
      StringTokenizer st = new StringTokenizer(message);

      return st.countTokens();
   }

/**
* Description: Gets the specific argument in a message.<P>
* Precondition: A message of type String must be sent.  Also, the
*     specific argument that the user wants to extract must be sent.
So,
*     if you want the 2nd word, then send the number 2.<P>
* Postcondition: The word that the user requested is returned.  If the
*     word does not exist (i.e. There is no second word/argument) then
*     the function returns null.
**/
   private static String getArgument(String message, int argNum) {
      int count = 1;
      StringTokenizer st = new StringTokenizer(message);

      while (st.hasMoreTokens()) {
            if (count == argNum)
                  return st.nextToken();

            count++;
            st.nextToken();
      }

      return null;
   }

   private int numCommand = 11;     //The number of mapped commands
   private String [] commandArray = new String[numCommand];
                                    //The mapped commands array
   private DataTurbine dt = new DataTurbine();
                                    //Establish connection to DT
   private DataTurbine dt2 = new DataTurbine();
                                    //Establish connection to DT
}
```

SectionC2: Serial Power Strip Echo (Client-Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 2/18/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class is used for receiving all information coming
 * from the Serial Power Strip located at the Ground Station.  This
 * class uses RBNB Data Turbine to establish a connection with the
 * ground station located at Santa Clara University Engineering
 * Department. For more information on DataTurbine and it's
 * functionality, please refer to the RBNB website:
 * http://rbnb.creare.com
 **/

import java.io.*;
import java.util.*;
import java.lang.*;

public class PowerStripClientEcho
{
	public static void main(String[] args)
	{
		DataTurbine dt = new DataTurbine();
			//DataTurbine Class used for connection
		String host;
			//Variable used to specify where DT Host is

		System.out.println("RACE: Power Strip Response Window\n");
		if(args.length>0)
		{
			host=args[0];
				//user must specify where DT Host is
			System.out.println("Connecting to " + host + "...");
		}
		else
		{
			System.out.println("ERROR: No RBNB host defined");
			return;
		}

				//Establish connection with Ground Station by
setting up channels
		dt.openPluginConnection(host, "receivePowerStrip");

		dt.addChannelReceive("text");
		dt.registerReceiveChannel();

		System.out.println("Connected...");
```

```
            while (true)
            {
                  String message = dt.receiveMessage(-1);
                        //blocking wait for information

                  if (message != null) {
                        System.out.println(message);
                              //if there's a message, output it
                        System.out.println();
                  }
            }
      }
}
```

Section C3: Serial Power Strip Server (Server-Side)

```
/*
 * @Author: Peter Salas
 *   <A HREF="mailto:PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * @version
 * Created: 2/23/2004
 *
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 *
 * Description: This class takes request coming in through DT and
 * forwards the command out to the Serial Power Strip that is connected
 * to the Serial Port. any response coming from the Serial Power Strip
 * is forwarded back out through
 * RBNB Data Turbine.
 *
 * @(#)TNCDT.java 1.12 98/06/25 SMI
 *
 * Copyright (c) 1998 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license
 * to use, modify and redistribute this software in source and binary
 * code form, provided that i) this copyright notice and license appear
 * on all copies of the software; and ii) Licensee does not utilize the
 * software in a manner which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind.
 * ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES,
 * INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND
 * ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY
 * LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE
 * SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS
 * BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES,
 * HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING
 * OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control
 * of aircraft, air traffic, aircraft navigation or aircraft
 * communications; or in the design, construction, operation or
 * maintenance of any nuclear facility. Licensee represents and
 * warrants that it will not use or redistribute the Software for such
 * purposes.
 */

import java.io.*;
import java.util.*;
import javax.comm.*;

public class PowerStripServer implements Runnable,
SerialPortEventListener {
      static CommPortIdentifier portId;
```

```
static Enumeration portList;
static String messageString = "";

static int stx = 0;
static boolean outputBufferEmptyFlag = false;

static InputStream      inputStream;
static OutputStream     outputStream;
static SerialPort serialPort;
Thread readThread;

static boolean portfound = false;
static boolean dataReady = false;
static String sBuffer = "";
static int num;

  public static void main(String[] args) {
        DataTurbine powerStripDT = new DataTurbine();
        String host;      //variable to store address of host
        String plugInName = "sendPowerStrip";
              // name of the plugin that will be used
              // to read information from DT.
        String sendChannel = "/sattest/receivePowerStrip/text";
              // the location of the channel
              // to write information to DT.
        String commport = "COM3";
              // the desired serial port to communicate on

        System.out.println("Welcome to the Power Strip to Data
Turbine program!");
        if(args.length>0){
              host=args[0];
        System.out.println("Connecting to " + host + "...");
        }
        else {
              System.out.println("ERROR: No RBNB host defined");
        return;
        }

  // plugin used to create a channel that you will read from when
  //client writes to channel
        powerStripDT.openPluginConnection(host,plugInName);
        powerStripDT.addChannelReceive("text");
        powerStripDT.registerReceiveChannel();

  // sink used to write a channel

  powerStripDT.openSinkConnection(host,"PowerStripServerSink");
        powerStripDT.addChannelSend(sendChannel);

        System.out.println("Connected...");

  //Search for COM3 port
        portList = CommPortIdentifier.getPortIdentifiers();

        while (portList.hasMoreElements()) {
```

```
                        portId = (CommPortIdentifier)
portList.nextElement();
                        if (portId.getPortType() ==
CommPortIdentifier.PORT_SERIAL) {
                        if (portId.getName().equals(commport)) {
                                    PowerStripServer powerstrip = new
PowerStripServer();
                              System.out.println(portId.getName() + "
found!");
                              portfound = true;
                        }
                        }
            }

            if (portfound == false)
            {
                  System.out.println("Port was not found...\n\nExiting
program....");
            }

            while(portfound){
                  if(dataReady){
                        try {
                        Thread.sleep(200);//ms wait to finish sending
data
                        // Push data onto the server:

                              //check if there is something in buffer
                        if (sBuffer != null && sBuffer != "")
                        {
                              System.out.println("Placing string: " +
sBuffer + " into server.");
                              powerStripDT.sendMessage(sBuffer);
                        }

                              //reset flags
                        dataReady=false;
                        sBuffer="";

                        } catch (InterruptedException e) {}
                  }

                        //send the message through DT
                  messageString = powerStripDT.receiveMessage();

                  if (messageString != null)
                  {
                        stx=1;//clear buffer
                        System.out.println("Data ready to be received
from DT!");

                        System.out.print("Command: " + messageString);
                        messageString = messageString + (char) 0xD;
                        String accept = "Y" + (char) 0xD;

                        try {
```

```
  outputStream.write(messageString.getBytes());
                  } catch (IOException e) {}

                  System.out.println(" sent to serial...\n");

                  try {
                        Thread.sleep(200);
                  } catch (Exception e) {}

                  try{
                        outputStream.write(accept.getBytes());
                  } catch (IOException e) {}


                  try {
                        Thread.sleep(200);
                  } catch (Exception e) {}

                  stx = 0;
            }
      }
}

public PowerStripServer() {
    try {
        serialPort = (SerialPort) portId.open("TNCDTApp", 2000);
    } catch (PortInUseException e) {}
    try {
        inputStream = serialPort.getInputStream();
              outputStream = serialPort.getOutputStream();
    } catch (IOException e) {}
  try {
        serialPort.addEventListener(this);
        } catch (TooManyListenersException e) {}
    serialPort.notifyOnDataAvailable(true);
    try {
                  serialPort.setSerialPortParams(9600,
            SerialPort.DATABITS_8,
            SerialPort.STOPBITS_1,
            SerialPort.PARITY_NONE);
    } catch (UnsupportedCommOperationException e) {}
    readThread = new Thread(this);
    readThread.start();
}

public void run() {
    try {
        Thread.sleep(20000);
    } catch (InterruptedException e) {}
}

public void serialEvent(SerialPortEvent event) {
    switch(event.getEventType()) {
    case SerialPortEvent.BI:
    case SerialPortEvent.OE:
    case SerialPortEvent.FE:
```

```
        case SerialPortEvent.PE:
        case SerialPortEvent.CD:
        case SerialPortEvent.CTS:
        case SerialPortEvent.DSR:
        case SerialPortEvent.RI:
        case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
            break;
        case SerialPortEvent.DATA_AVAILABLE:
            byte[] readBuffer = new byte[24];

           try {
                while (inputStream.available() > 0) {
                    int numBytes = inputStream.read(readBuffer);
                }
                        if(stx==0){
                              dataReady=true;
                              sBuffer = sBuffer + new
String(readBuffer);
                        }

            } catch (IOException e) {}
            break;
        }
    }
}
```

Appendix D

Section D1: Predict Interpreter (Client-Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 2/20/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class is used for validating user inputs, and
 * interpreting inputs to the actual commands that the Ground Station
 * accepts for remote access. Specifically, this class validates
 * commands for Predict, the Satellite Tracker Program. To esblish a
 * connection with the Ground Station located at Santa Clara University
 * Engineering Department. For more information on DataTurbine
 * and it's functionality, please refer to the RBNB website:
 * http://rbnb.creare.com
 **/

import java.io.*;
import java.util.*;

public class PredictClientInterpreter {

/*********************************************************************/
/*********************        Constructor       *********************/
/*********************************************************************/

/**
* Description: initializes the mapped variables for the control of
*     Predict (Satellite Tracker). This constructor also establishes a
*     connection to the Ground Station using DT, so that it can
*     send commands to the device.<P>
* Precondition: The IP of the DT Server must be given to establish a
*     connection<P>
* Postcondition: all variables are initialized to its default value. If
*     a connection was made successfully to the DT Server, this
*     constructor will return with no errors.  If there is
*     an error connecting, then the constructor will return with an
*     RBNB DataTurbine error.
**/
   public PredictClientInterpreter(String host) {
      commandArray[0] = "predict -update";
      commandArray[1] = "predict -poss";
      commandArray[2] = "predict -pass";
      commandArray[3] = "predict -help";
```

```
      dt.openSinkConnection(host,"PredictClientSink");
      dt.addChannelSend("/sattest/sendPredict/text");
   }

/**********************************************************************/
/******************          Accessors & Mutators          **********/
/**********************************************************************/

/**
* Description: This function validates a user command.<P>
* Precondition: The user input must be sent of type String.<P>
* Postcondition: If the command is a valide command, this function
*     return true. Else, it returns false
**/
   public boolean validateCommand(String command) {
      return (getCommandIndex(command) != -1);
   }

/**
* Description: This sends the command to the Ground Station<P>
* Precondition: The command from the user must be sent of type
*     String<P>
* Postcondition:  If the command is a valid command, then the command
*     is sent and function returns true. Else, the function returns
*     false.
**/
   public boolean sendCommand(String command, int wait) {
      int index = getCommandIndex(command);

      switch (index) {
            case 0: if (countArguments(command) > 2) {
                        String message = getArgument(command, 2) + " "
+ getArgument(command, 3);
                        dt.sendMessage(message, wait);
                        return true;
                  }

                  return false;
            case 1: if (countArguments(command) > 2) {
                        String message = getArgument(command, 2) + " "
+ getArgument(command, 3);
                        dt.sendMessage(message, wait);
                        return true;
                  }

                  return false;
            case 2: if (countArguments(command) > 2) {
                        String message = getArgument(command, 2) + " "
+ getArgument(command, 3);
                        dt.sendMessage(message, wait);
                        return true;
                  }

                  return false;
            case 3: dt.sendMessage("-help", wait);
                  return true;
```

```
            default: return false;
      }
   }

/**
* Description: This goes through the mapped commands of Predict, and
*     returns the index of that command.<P>
* Precondition: The user command must be sent of type String.<P>
* Postcondition: If the command is valid, then this function returns
*     the index of the command. If the command is not valid, this
*     function returns -1.
**/
   private int getCommandIndex(String command) {
      for(int i=0; i<numCommand; i++) {
            if (command.toLowerCase().startsWith(commandArray[i]))
                  return i;
      }

      return -1;
   }

/**
* Description: This counts the number of arguments in a message of type
*     String<P>
* Precondition: A message of type String must be sent<P>
* Postcondition: the number of words in a message is returned<P>
* NOTE: Words are deliminated by spaces.
**/
   private static int countArguments(String message) {
      StringTokenizer st = new StringTokenizer(message);

      return st.countTokens();
   }

/**
* Description: Gets the specific argument in a message.<P>
* Precondition: A message of type String must be sent.  Also, the
*     specific argument that the user wants to extract must be sent.
So,
*     if you want the 2nd word, then send the number 2.<P>
* Postcondition: The word that the user requested is returned.  If the
*     word does not exist (i.e. There is no second word/argument) then
*     the function returns null.
**/
   private static String getArgument(String message, int argNum) {
      int count = 1;
      StringTokenizer st = new StringTokenizer(message);

      while (st.hasMoreTokens()) {
            if (count == argNum)
                  return st.nextToken();

            count++;
            st.nextToken();
      }

      return null;
```

```
    }

    private int numCommand = 4;       //The number of commands
    private String [] commandArray = new String[numCommand];
                                      //mapped command array
    private DataTurbine dt = new DataTurbine();
                                      //Establish connection to DT
}
```

Section D2: Predict Echo (Client-Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 2/18/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class is used for receiving all information coming
 * from the Predict Program (Satellite Tracker) located at the Ground
 * Station.  This class uses RBNB Data Turbine to establish a
 * connection with the ground station located at Santa Clara University
 * Engineering Department. For more information on DataTurbine and it's
 * functionality, please refer to the RBNB website:
 * http://rbnb.creare.com
 **/

import java.io.*;
import java.util.*;
import java.lang.*;

public class PredictClientEcho
{
      public static void main(String[] args)
      {
            DataTurbine dt = new DataTurbine();
                  //DataTurbine Class used for connection
            String host;
                  //Variable used to specify where DT Host is

            System.out.println("RACE: Predict Response Window\n");
            if(args.length>0)
            {
                  host=args[0];
                        //user must specify where DT Host is
                  System.out.println("Connecting to " + host + "...");
            }
            else
            {
                  System.out.println("ERROR: No RBNB host defined");
                  return;
            }

      //Establish connection with Ground Station by setting up channels
            dt.openPluginConnection(host, "receivePredict");

            dt.addChannelReceive("text");
            dt.registerReceiveChannel();

            System.out.println("Connected...");
```

```
			while (true)
			{
				String message = dt.receiveMessage(-1);
					//blocking wait for information

				if (message != null) {
					System.out.println(message);
						//if there's a message, output it
					System.out.println();
				}
			}
		}
	}
```

Section D3: Predict Server (Server-Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 5/14/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class takes coordinates the actions and requests
 * from the RBNB DataTurbine.  Specifically, this forwards all commands
 * coming in from DT to the PredictServerInterpreter, which interprets
 * the user inputs.  All, responses are then forwarded back to the user
 * through DT.
 **/

import java.io.*;
import java.util.*;

public class PredictServer {

   public static void main(String[] args) {
      PredictServerInterpreter predictInterpreter = new
PredictServerInterpreter();

      DataTurbine predictDT = new DataTurbine();
      String host;            //variable to store address of host
      String plugInName = "sendPredict";
                        // name of the plugin that will be used
                        // to read information from DT.
      String sendChannel = "/sattest/receivePredict/text";
                  // the location of the channel
                  // to write information to DT.

      int countArgs=0;  //The number of arguments that come in from DT
      String dtMessage; //The message from DT
      String command;         //The command portion of the dtMessage
      String subCommand;      //The subcommand portion of the dtMessage
      String output;          //The response from the Predict Program
      String errorMsg = "ERROR: Invalid Command \n--Type \"-help\" for
more information on how to send commands to Predict Software";
            //Error Message if an invalid command from the user

      System.out.println("Welcome to the Predict to Data Turbine
program!");

      if(args.length>0){
            host=args[0];
                  //user must specify the IP of the DT Server
            System.out.println("Connecting to " + host + "...");
      }
```

```
      else {
            System.out.println("ERROR: No RBNB host defined");
            return;
      }

            //Establishes a connection to the DT Server
      predictDT.openPluginConnection(host,plugInName);
      predictDT.addChannelReceive("text");
      predictDT.registerReceiveChannel();

      predictDT.openSinkConnection(host,"PredictServerSink");
      predictDT.addChannelSend(sendChannel);

      System.out.println("Connected...");

      while (true) {
            dtMessage = predictDT.receiveMessage(-1); //Blocking
request for information on DT

                  //Break the message up into parts
            countArgs = PredictServer.countArguments(dtMessage);
            command = PredictServer.getArgument(dtMessage, 1);
            subCommand = PredictServer.getArgument(dtMessage, 2);

            if (predictInterpreter.validateCommand(command)) {
                  System.out.println("validate Worked\n");
                        //perform command and save the output
                  if (subCommand == null)
                        output =
predictInterpreter.performCommand(command,"");
                  else
                        output =
predictInterpreter.performCommand(command,subCommand);
                        //send the output through DT back to the user
                  predictDT.sendMessage(output, 1);
            }
            else { predictDT.sendMessage(errorMsg, 1); }
      }


   }

/**
* Description: Counts the number of arguments<P>
* Precondition: The message of type String must be passes in<P>
* Postcondition: The number of arguments is returned
**/
   private static int countArguments(String message) {
      StringTokenizer st = new StringTokenizer(message);

      return st.countTokens();
   }

/**
* Description: This gets the specific argument in a message.<P>
* Precondition: There must be a message of type String passed in, and
*     also the specific argument that the user wants to "pull out." So
```

```
*     if the user wants the second argument, they would pass the number
*     "2" in.<P>
* Postcondition: If there is word at the specified position, that word
*     is returned.  If not, then the function returns null.
**/
   private static String getArgument(String message, int argNum) {
      int count = 1;
      StringTokenizer st = new StringTokenizer(message);

      while (st.hasMoreTokens()) {
            if (count == argNum)
                  return st.nextToken();

            count++;
            st.nextToken();
      }

      return null;
   }
}
```

Section D4: Predict (Server-Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 3/10/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class is used for communicating with the Predict
 * Program Predict is Dos-command based program, so this program writes
 * out to the command prompt and stores the information coming back
 * onto a buffer. This buffer is then returned back to the calling
 * function.  Further information on the predict program that we use,
 * please refer to this website
 *          <A HREF="http://www.qsl.net/kd2bd/predict.html">
 *                (http://www.qsl.net/kd2bd/predict.html)
 *          </A>
 **/

import java.lang.*;
import java.io.*;

public class Predict {

/**
* Description: This requests the current position of a particular
*     satellite. This class depends on the Predict program and the two
*     line keplerian element file (predict.tle).  If the satellite
*     requested is not in this list, then there will be no response.<P>
* Precondition: The satellite must be in the predict.tle file.<P>
* Postcondition: The information about the current position of the
*     satellite will be returned.
**/
      public static String getSatelliteInfo(String satellite) {
            try {
                  Process process = Runtime.getRuntime().exec("predict
-f " + satellite);;
                  process.waitFor();

                  InputStream in = process.getInputStream();

                  String stringIn = "";
                  int charIn;
                  char tempChar;

                  while( (charIn = in.read()) != -1 ) {
                        tempChar = (char)charIn;
                        Character tChar = new Character(tempChar);
                        stringIn += tChar.toString();
                  }
                  return stringIn;
```

```
		}
		catch(Exception e) { System.out.println(e); return null;}
	}

/**
* Description: This function requests an update of the predict.tle file
*	from
*	<A HREF="http_get http://www.qsl.net/kd2bd/predict.tle">
*		http_get http://www.qsl.net/kd2bd/predict.tle
*	</A>.<P>
* Precondition: The tle file that you want to update must be given of
*	type String RACE is specifically using the "predict.tle" file, so
*	this must be the parameter passed.<P>
* Postcondition: The function will return "Update Successful" if there
*	is no error.
**/
	public static boolean updateKeplerians(String tleFile)
	{
		try {
			Process process1 =
Runtime.getRuntime().exec("http_get
http://www.qsl.net/kd2bd/predict.tle " + tleFile);
			process1.waitFor();

	System.out.println("Predict.updateKeplerians(tleFile)--
>Downloading update");
			Process process2 = Runtime.getRuntime().exec("predict
-u " + tleFile);;
			process2.waitFor();

	System.out.println("Predict.updateKeplerians(tleFile)-->Update
Completed");
			return true;
		}
		catch(Exception e) { System.out.println(e); return false;}
	}

/**
* Description: This function requests the next pass of a specific
*	satellite.  Like the getSatelliteInfo() function, this function
*	depends on the predict.tle file If the satellite is not in the
*	list, then this function does not return anything.<P>
* Precondition: The satellite must be in the predict.tle file<P>
* Postcondition: Information about when and where the next pass will be
*	is returned.
**/
	public static String getNextPass(String satellite) {
		try {
			Process process = Runtime.getRuntime().exec("predict
-p " + satellite);;
			process.waitFor();

			InputStream in = process.getInputStream();

			String stringIn = "";
			int charIn;
			char tempChar;
```

```
                        while( (charIn = in.read()) != -1 ) {
                                tempChar = (char)charIn;
                                Character tChar = new Character(tempChar);
                                stringIn += tChar.toString();
                        }
                        return stringIn;
                }
                catch(Exception e) { System.out.println(e); return null;}
        }

}
```

Section D5: Predict Interpreter (Server-Side)

```
/**
 * @Author: Peter Salas
 * <A HREF="mailto:PSalas@scu.edu"> (Peter Salas) </A>
 *
 * @Description: This class is responsible for validating commands for
 * the program Predict.  If a valid command, then this class processes
 * the request and returns if the command was performed successfully.
 *
 */
import java.io.*;
import java.util.*;

public class PredictServerInterpreter {

/********************************************************************/
/******************           Constructor       *********************/
/********************************************************************/

/**
* Description: initializes the mapped variables for the control of the
*     Predict program.
* Precondition: No Precondition<P>
* Postcondition: all variables are initialized to its default value.
**/
   public PredictServerInterpreter() {
      commandArray[0] = "-update";
      commandArray[1] = "-poss";
      commandArray[2] = "-pass";
      commandArray[3] = "-help";
   }

/********************************************************************/
/*****************            Accessors & Mutators          **********/
/********************************************************************/

/**
* Description: This function validates a user command.<P>
* Precondition: The user input must be sent of type String.<P>
* Postcondition: If the command is a valide command, this function
*     return true. Else, it returns false
**/
   public boolean validateCommand(String command) {
      return (getCommandIndex(command) != -1);
   }

/**
* Description: This sends the command to the Predict program<P>
* Precondition: The command from the user must be sent of type
*     String<P>
* Postcondition:  If the command is a valid command, then the command
*     is sent and function returns the information coming from the
*     Program. Else, the function returns null.
**/
```

```
   public String performCommand(String command, String subCommand) {
      int commandIndex = getCommandIndex(command);

      System.out.println("\"" + command +"\"");
      System.out.println("\"" + subCommand +"\"");

      switch (commandIndex) {
            case 0: if (Predict.updateKeplerians(subCommand)) {
                        System.out.println("Update Completed");
                        return "Update Completed";
                    }
                  System.out.println("Update Unsuccessful");
                  return "Update Unsuccessful";

            case 1: System.out.println("Getting Satellite Info");
                    return Predict.getSatelliteInfo(subCommand);

            case 2: System.out.println("Getting next Pass of
Satellite");
                    return Predict.getNextPass(subCommand);

            case 3: System.out.println("Getting Help");
                    return getHelp();

            default: return null;

      }
   }

/**
* Description: This goes through the mapped commands of the Predict
*     program, and returns the index of that command.<P>
* Precondition: The user command must be sent of type String.<P>
* Postcondition: If the command is valid, then this function returns
*     the index of the command. If the command is not valid, this
*     function returns -1.
**/
   private int getCommandIndex(String command) {
      for(int i=0; i<numCommand; i++) {
            if (commandArray[i].equalsIgnoreCase(command))
                  return i;
      }

      return -1;
   }

/**
* Description: This outputs the help menu<P>
* Precondition: No Conditions<P>
* Postcondition: The function returns a String help message.
**/
private String getHelp() {
      String help = "Help File for Predict\n--------------------
\n\ncommands:\n   -update <.tle file containing Keplerians>\n\tNOTE:
the kep file you should update should be predict.tle\n\n   -poss
<satellite>\n\tNOTE: If you are trying to talk to Sappire, enter in
\"OSCAR-45\" as the satellite\n\n   -pass <satellite>\n\n   -help";
```

```
      return help;
   }

   private int numCommand = 4;
   private String [] commandArray = new String[numCommand];
}
```

Appendix E

Section E1: Antenna Interpreter (Client-Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 2/20/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class is used for validating user inputs, and
 * interpreting inputs to the actual commands that the Ground Station
 * accepts for remote access. Specifically, this class validates
 * commands for Labjack (Antenna Controller). To establish a connection
 * with the Ground Station located at Santa Clara University
 * Engineering Department. For more information on DataTurbine
 * and it's functionality, please refer to the RBNB website:
 * http://rbnb.creare.com
 **/

import java.io.*;
import java.util.*;

public class AntennaClientInterpreter {

/*******************************************************************/
/******************          Constructor       *****************/
/*******************************************************************/

/**
* Description: initializes the mapped variables for the control of the
*     Labjack (Antenna Controler). This constructor also establishes a
*     connection to the Ground Station using DT, so that it can
*     send commands to the device.<P>
* Precondition: The IP of the DT Server must be given to establish a
*     connection<P>
* Postcondition: all variables are initialized to its default value. If
*     a connection was made successfully to the DT Server, this
*     constructor will return with no errors.  If there is
*     an error connecting, then the constructor will return with an
*     RBNB DataTurbine error.
**/
   public AntennaClientInterpreter(String host) {
      commandArray[0] = "antenna -t";
      commandArray[1] = "antenna -a";
      commandArray[2] = "antenna -s";
      commandArray[3] = "antenna -help";

      dt.openSinkConnection(host,"AntennaClientSink");
```

```
      dt.addChannelSend("/sattest/sendAntenna/text");
   }

/****************************************************************/
/***************       Accessors & Mutators         ***************/
/****************************************************************/

/**
* Description: This function validates a user command.<P>
* Precondition: The user input must be sent of type String.<P>
* Postcondition: If the command is a valid command, this function
*     return true. Else, it returns false
**/
   public boolean validateCommand(String command) {
      return (getCommandIndex(command) != -1);
   }

/**
* Description: This sends the command to the Ground Station<P>
* Precondition: The command from the user must be sent of type
*     String<P>
* Postcondition:  If the command is a valid command, then the command
*     is sent and function returns true. Else, the function returns
*     false.
**/
   public boolean sendCommand(String command, int wait) {
      int index = getCommandIndex(command);

      switch (index) {
            case 0: dt.sendMessage("-t",wait);
                    return true;

            case 1: dt.sendMessage("-a",wait);
                    return true;

            case 2: dt.sendMessage("-s",wait);
                    return true;

            case 3: dt.sendMessage("-help",wait);
                  return true;

            default: return false;
      }
   }

/**
* Description: This goes through the mapped commands of the Labjack
*     program, and returns the index of that command.<P>
* Precondition: The user command must be sent of type String.<P>
* Postcondition: If the command is valid, then this function returns
*     the index of the command. If the command is not valid, this
*     function returns -1.
**/
   private int getCommandIndex(String command) {
      for(int i=0; i<numCommand; i++) {
            if (commandArray[i].equalsIgnoreCase(command))
                  return i;
```

```
      }

      return -1;
   }

   private int numCommand = 4;             //The number of mapped
commands
   private String [] commandArray = new String[numCommand];
            //The array of commands
   private DataTurbine dt = new DataTurbine();
            //DataTurbine class used for establishing connection
}
```

Section E2: Antenna Echo (Client-Side)

```
/**
 * @Author: Peter Salas
 *    <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 * <P>
 * @version
 * Created: 5/12/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team <P>
 * Description: This class is used for receiving all information coming
 * from the Labjack, which is connected to the antenna.  This class
 * uses RBNB Data Turbine to establish a connection with the ground
 * station located at Santa Clara University Engineering Department.
 * For more information on DataTurbine and its functionality, please
 * refer to the RBNB website: http://rbnb.creare.com
 **/

import java.io.*;
import java.util.*;
import java.lang.*;

public class AntennaClientEcho
{
	public static void main(String[] args)
	{
		DataTurbine dt = new DataTurbine();
			//DataTurbine Class used for connection
		String host;
	//Variable used to specify where DT Host is

		System.out.println("RACE: Antenna Response Window\n");
		if(args.length>0)
		{
			host=args[0];
				//user must specify where DT Host is
			System.out.println("Connecting to " + host + "...");
		}
		else
		{
			System.out.println("ERROR: No RBNB host defined");
			return;
		}

	//Establish connection with Ground Station by setting up channels
		dt.openPluginConnection(host, "receiveAntenna");

		dt.addChannelReceive("text");
		dt.registerReceiveChannel();

		System.out.println("Connected...");

		while (true)
		{
			String message = dt.receiveMessage(-1);
```

```
                        //blocking wait for information

                  if (message != null) {
                        System.out.println(message);
                              //if there's a message, output it
                        System.out.println();
                  }
            }
      }
}
```

Section E3: Antenna Server (Server-Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 5/14/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class takes coordinates the actions and requests
 * from the RBNB DataTurbine.  Specifically, this forwards all commands
 * coming in from DT to the AntennaServerInterpreter, which interprets
 * the user inputs.  All, responses are then forwarded back to the user
 * through DT.
 **/

import java.io.*;
import java.util.*;

public class AntennaServer {

   public static void main(String[] args) {
      AntennaServerInterpreter antennaInterpreter = new
AntennaServerInterpreter();

      DataTurbine antennaDT = new DataTurbine();
      String host;            //variable to store address of host
      String plugInName = "sendAntenna";
                  // name of the plugin that will be used
                  // to read information from DT.
      String sendChannel = "/sattest/receiveAntenna/text";
                  // the location of the channel
                  // to write information to DT.

      String dtMessage;       //The message coming in from DT
      String output;    //The message that will be written out to DT
      String errorMsg = "ERROR: Invalid Command \n--Type \"-help\" for
more information on how to send commands to Antenna Software";


      System.out.println("Welcome to the Antenna to Data Turbine Server
program!");

      if(args.length>0){
            host=args[0];
                  //The user must enter in the IP of the DT Server
            System.out.println("Connecting to " + host + "...");
      }

      else {
            System.out.println("ERROR: No RBNB host defined");
            return;
```

```
      }


//Establish a connection with the DT Server and setup Channels
      antennaDT.openPluginConnection(host,plugInName);
      antennaDT.addChannelReceive("text");
      antennaDT.registerReceiveChannel();

      antennaDT.openSinkConnection(host,"AntennaServerSink");
      antennaDT.addChannelSend(sendChannel);

      System.out.println("Connected...");

      while (true) {
            dtMessage = antennaDT.receiveMessage(-1);
                  //blocking call to receive data from DT

            if (antennaInterpreter.validateCommand(dtMessage)) {
                  output =
antennaInterpreter.performCommand(dtMessage);

                  antennaDT.sendMessage(output, 1);
                        //wait only 1 milisecond for a response
            }
            else { antennaDT.sendMessage(errorMsg, 1); }
      }


   }
}
```

Section E4: Antenna (Server-Side)

```
/**
 * @Author: Peter Salas
 *    <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 5/14/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class is used for communicating with the
 * AntennaAutoTracker program that was written by Graduate Student, Dan
 * Schuet, of the Santa Clara University Engineering Department. This
 * program combines the Predict program and the Labjack program
 * to provide control functionality of the Antenna.  This program
 * allows the user to ask for the current location of the Antenna and
 * also tell the antenna to autotrack to the SAPPHIRE Satellite (The
 * Stanford Satellite, a.k.a OSCAR-45, NO-45).
 **/

import java.lang.*;
import java.io.*;

public class Antenna {

/*****************************************************************/
/****************          Accessors         ********************/
/*****************************************************************/

/**
* Description: Sends a command to AntennaAutoTracker, telling it to
*     autotrack a Satellite.  Currently, this program is set to
*     autotrack to SAPPHIRE (The Stanford Satellite), if you want it to
*     track a different satellite, then you will have to change the C
*     code for predict.<P>
* Precondition: The Antenna Controller must be connected to labjack and
*     receiving power<P>
* Postcondition: This function will passes the command to another
*     program, and that program will execute in its own thread.
*     Therefore, this is a non-blocking message passing to turn the
*     antenna.
**/
      public static boolean autoTrackSatellite() {
            try {
                  Process process =
Runtime.getRuntime().exec("AntennaAutoTracker -t");

                  return true;
            }
            catch(Exception e) { System.out.println(e); return false;}
      }

/**
```

```
* Description: Sends a command to AntennaAutoTracker asking it where
*     the antenna is pointed now.<P>
* Precondition: The Antenna Controller must be connected to labjack and
*     receiving power<P>
* Postcondition: The function returns the current position of the
*     antenna.
**/
      public static String getAntennaPosition()
      {
            try {
                  Process process =
Runtime.getRuntime().exec("AntennaAutoTracker -a");
                  process.waitFor();

                  InputStream in = process.getInputStream();

                  String stringIn = "";
                  int charIn;
                  char tempChar;

                  while( (charIn = in.read()) != -1 ) {
                        tempChar = (char)charIn;
                        Character tChar = new Character(tempChar);
                        stringIn += tChar.toString();
                  }
                  return stringIn;
            }
            catch(Exception e) { System.out.println(e); return null;}
      }

/**
* Description: Sends a command to AntennaAutoTracker asking it where
*     SAPPHIRE is currently<P>
* Precondition: The Antenna Controller must be connected to labjack and
*     receiving power<P>
* Postcondition: The function returns the current position of SAPPHIRE.
**/
      public static String getSatellitePosition() {
            try {
                  Process process =
Runtime.getRuntime().exec("AntennaAutoTracker -s");;
                  process.waitFor();

                  InputStream in = process.getInputStream();

                  String stringIn = "";
                  int charIn;
                  char tempChar;

                  while( (charIn = in.read()) != -1 ) {
                        tempChar = (char)charIn;
                        Character tChar = new Character(tempChar);
                        stringIn += tChar.toString();
                  }
                  return stringIn;
            }
            catch(Exception e) { System.out.println(e); return null;}
```

```
      }

/**
* Description: This is the help menu for the AntennaAutoTracker<P>
* Precondition: AntennaAutoTracker must be within same directory as
*     this class file<P>
* Postcondition: The help menu is returned to user.
**/
      public static String getHelp() {
            try {
                  Process process =
Runtime.getRuntime().exec("AntennaAutoTracker");;
                  process.waitFor();

                  InputStream in = process.getInputStream();

                  String stringIn = "";
                  int charIn;
                  char tempChar;

                  while( (charIn = in.read()) != -1 ) {
                        tempChar = (char)charIn;
                        Character tChar = new Character(tempChar);
                        stringIn += tChar.toString();
                  }
                  return stringIn;
            }
            catch(Exception e) { System.out.println(e); return null;}
      }

}
```

Section E5: Antenna Interpreter (Server-Side)

```
/**
 * @Author: Peter Salas
 *   <A HREF="mailto: PSalas@scu.edu"> (PSalas@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 5/14/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class is responsible for taking in user commands
 * and (1) validating the command, and if valid (2) perform the
command.
 * Once the command is performed, then the program returns the response
 * from the Antenna Program.
 *
 * NOTE: This program works with the AntennaServer class and the
 * Antenna class.  The AntennaServer class is responsible for taking in
 * the command from the user and forwarding the response back to the
 * user.  The Antenna class is responsible for sending the command to
 * the AntennaControler device/program.
 **/

import java.io.*;
import java.util.*;

public class AntennaServerInterpreter {

/*********************************************************************/
/********************         Constructor       *********************/
/*********************************************************************/

/**
* Description: initializes the mapped variables for the control of the
*     Labjack (Antenna Controler). This constructor also establishes a
*     connection to the Ground Station using DT, so that it can send
*     commands to the device.<P>
* Precondition: No Precondition<P>
* Postcondition: all variables are initialized to its default value. If
*     a connection was made successfully to the DT Server, this
*     constructor will return with no errors.  If there is an error
*     connecting, then the constructor will return with an RBNB
*     DataTurbine error.
**/
   public AntennaServerInterpreter() {
      commandArray[0] = "-t";       //auto tracking
      commandArray[1] = "-a";       //antenna orientation
      commandArray[2] = "-s";       //satellite position
      commandArray[3] = "-help";

      antennaDT = new DataTurbine();
                  //Establish connection to DT Server located at Santa
Clara University Ground Station
```

```
                  //This should change if the DT Server moves
      antennaDT.openSinkConnection("localhost:3333",
"AntennaInterpreterSink");
      antennaDT.addChannelSend("/sattest/receiveAntenna/text");
   }

/**********************************************************************/
/******************          Accessors & Mutators          **********/
/**********************************************************************/

/**
* Description: This function validates a user command.<P>
* Precondition: The user input must be sent of type String.<P>
* Postcondition: If the command is a valide command, this function
*     return true. Else, it returns false
**/
   public boolean validateCommand(String command) {
      return (getCommandIndex(command) != -1);
   }

/**
* Description: This sends the command to the AntennaAutoTracker
*     program<P>
* Precondition: The command from the user must be sent of type
*     String<P>
* Postcondition:  If the command is a valid command, then the command
*     is sent and function returns the information coming from the
*     Program. Else, the function returns null.
**/
   public String performCommand(String command) {
      int commandIndex = getCommandIndex(command);

      switch (commandIndex) {
            case 0: antennaDT.sendMessage("Auto Tracking Started...",
1);
                    Antenna.autoTrackSatellite();

            case 1: return Antenna.getAntennaPosition();

            case 2: return Antenna.getSatellitePosition();

            case 3: return Antenna.getHelp();

            default: return null;

      }
   }

/**
* Description: This goes through the mapped commands of the Labjack
*     program, and returns the index of that command.<P>
* Precondition: The user command must be sent of type String.<P>
* Postcondition: If the command is valid, then this function returns
*     the index of the command. If the command is not valid, this
*     function returns -1.
**/
   private int getCommandIndex(String command) {
```

```
        for(int i=0; i<numCommand; i++) {
              if (commandArray[i].equalsIgnoreCase(command))
                    return i;
        }

        return -1;
    }

    private int numCommand = 4;             //The number of commands
    private String [] commandArray = new String[numCommand];
                                      //The command array
    private DataTurbine antennaDT;
                    //Establish a connection to DT Server
}
```

Section E6: AntennaAutoTracker (Server-Side)

```
//*********************************************************************
//
// File Name    : 'AntennaAutoTracker.c'
// Author       : Daniel Schuet - Copyright (C) 2004
// Created      : 2004-05
// Version      : 1.0
// Description  : Automatically controls an antenna to track a
// satellite in the sky. Uses a G-5500 Yaesu Rotor along with
// LabJack.  Uses Predict software to get satellite azimuth and
// elevation.
//
//*********************************************************************

// *** Needs labjack library to build (ljackuw.lib)

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <wtypes.h>
#include "ljackuw.h"

// CALIBRATION PARAMETERS
#define EL_MIN_DEG      0.0
#define EL_MIN_AD 0.0         // analog voltage from rotor

#define EL_MAX_DEG      180.0
#define EL_MAX_AD 247.0

#define AZ_MIN_DEG      0.0
#define AZ_MIN_AD 25.0

#define AZ_MAX_DEG      360.0
#define AZ_MAX_AD 213.0

#define THRESH_DEG      2.0   // +/- 2 degress

// Labjack Channels
#define EL_UP_CHANNEL   2
#define EL_DN_CHANNEL   3
#define AZ_POS_CHANNEL  0
#define AZ_NEG_CHANNEL  1

// Func. Prototypes

//! This function gives you the azimuth and elevation of satellite
using Predict.
//  Returns true if satellite is currently in view, and false if it is
not.
bool SatPosition(int *output_az, int *output_el);

//! Parse through what Predict returns and get azimuth and elevation.
void parsePredictLine(char * buffer, int *AZ, int *EL);
```

```
//! Finds and returns azimuth for next satellite pass.
int  getNextPassAZ(void);

//! Reads the current antenna azimuth and elevation.
void readCurrentAntennaOrientation(float *AZ, float *EL);

//! Antenna Motion Functions
void antennaMoveUp(void);
void antennaMoveDown(void);
void antennaRotatePos(void);
void antennaRotateNeg(void);
void antennaSTOPEL(void);
void antennaSTOPAZ(void);
void antennaSTOPALL(void);

void main(int nargs, char* args[])
{
      int npAZ;
      int npEL=0;

      int sat_currAZ;
      int sat_currEL;

      bool again = true;
      bool cont = true;

      float ant_currAZ, ant_currEL, errorEL, errorAZ;

      // help
      if(nargs==1)
      {
            printf("AntennaAutoTracker [options]\n");
            printf("                    -t, track\n");
            printf("                    -a, antenna orientation\n");
            printf("                    -s, satellite position\n");
            exit(1);
      }

      else if(nargs>1)
      {
            // antenna orientation
            if(strcmp(args[1],"-a")==0)
            {

      readCurrentAntennaOrientation(&ant_currAZ,&ant_currEL);

      printf("ANTENNA\nazimuth=%d\nelevation=%d\n",int(ant_currAZ+0.5),
int(ant_currEL+0.5));
            }
            // satellite position
            else if(strcmp(args[1],"-s")==0)
            {
                  SatPosition(&sat_currAZ, &sat_currEL);

      printf("SAPPHIRE\nazimuth=%d\nelevation=%d\n",sat_currAZ,sat_curr
EL);
            }
```

```
            // start autotracking
            else if(strcmp(args[1],"-t")==0)
            {
                  if(SatPosition(&sat_currAZ, &sat_currEL))
                  {
                        printf("Satellite is currently in view.\n");
                        printf("Autotracking...\n");

                        ::Sleep(5000);

                        while(SatPosition(&sat_currAZ, &sat_currEL) &&
!kbhit())
                        {
                              cont=true;
                              while(cont && !kbhit())
                              {
                                    ::Sleep(500);

                                    SatPosition(&sat_currAZ,
&sat_currEL);

      readCurrentAntennaOrientation(&ant_currAZ,&ant_currEL);

                                    errorEL = ant_currEL-
float(sat_currEL);
                                    errorAZ = ant_currAZ-
float(sat_currAZ);

                                    system("cls");
                                    printf("Satellite Az = %d \t\t
Satellite El = %d\n",sat_currAZ,sat_currEL);
                                    printf("Antenna   Az = %d \t\t
Antenna   El = %d\n",int(ant_currAZ+0.5),int(ant_currEL+0.5));
                                    printf("\nElevation error =
%f\n",errorEL);
                                    printf("Azimuth   error =
%f\n",errorAZ);


                                    if(errorEL > THRESH_DEG)
                                          antennaMoveDown();
                                    else if (errorEL < -THRESH_DEG)
                                          antennaMoveUp();

                                    if(errorAZ > THRESH_DEG)
                                          antennaRotateNeg();
                                    else if (errorAZ < -THRESH_DEG)
                                          antennaRotatePos();


                                    if(errorEL<THRESH_DEG && errorEL>-
THRESH_DEG)

                                          antennaSTOPEL();

                                    if(errorAZ<THRESH_DEG && errorAZ>-
THRESH_DEG)

                                          antennaSTOPAZ();
```

```
                                    if(errorEL<THRESH_DEG && errorEL>-
THRESH_DEG && errorAZ<THRESH_DEG && errorAZ>-THRESH_DEG)
                                    {
                                          antennaSTOPALL();
                                          cont=false;
                                    }
                              }
                        }
                  }

                  else
                  {
                        printf("\nSatellite is currently NOT in
view.\n");

                        npAZ = getNextPassAZ();

                        printf("\nMoving antenna to \n%d degrees
elevation and \n%d degrees azimuth \nfor next pass...\n", npEL, npAZ);

                        ::Sleep(5000);

                        // Motion Test code
                        /*printf("Moving up...\n");
                        antennaMoveUp();
                        ::Sleep(1000);
                        antennaSTOPALL();
                        ::Sleep(1000);

                        printf("Moving down...\n");
                        antennaMoveDown();
                        ::Sleep(1000);
                        antennaSTOPALL();
                        ::Sleep(1000);

                        printf("Rotating positive...\n");
                        antennaRotatePos();
                        ::Sleep(1000);
                        antennaSTOPALL();
                        ::Sleep(1000);

                        printf("Rotating negative...\n");
                        antennaRotateNeg();
                        ::Sleep(1000);
                        antennaSTOPALL();
                        ::Sleep(1000);*/

                        while(again && !kbhit()){
                              ::Sleep(100);

      readCurrentAntennaOrientation(&ant_currAZ,&ant_currEL);

                              errorEL = ant_currEL-float(npEL);
                              errorAZ = ant_currAZ-float(npAZ);

                              system("cls");
```

```
                              printf("Elevation Error = %f\n",errorEL);
                              printf("Azimuth Error   = %f\n",errorAZ);

                              if(errorEL > THRESH_DEG)
                                    antennaMoveDown();
                              else if (errorEL < -THRESH_DEG)
                                    antennaMoveUp();

                              if(errorAZ > THRESH_DEG)
                                    antennaRotateNeg();
                              else if (errorAZ < -THRESH_DEG)
                                    antennaRotatePos();


                              if(errorEL<THRESH_DEG && errorEL>-
THRESH_DEG)
                                    antennaSTOPEL();

                              if(errorAZ<THRESH_DEG && errorAZ>-
THRESH_DEG)
                                    antennaSTOPAZ();

                              if(errorEL<THRESH_DEG && errorEL>-
THRESH_DEG && errorAZ<THRESH_DEG && errorAZ>-THRESH_DEG)
                              {
                                    antennaSTOPALL();
                                    again=false;
                              }
                        }

                        printf("Done.\n");
                  }

                  antennaSTOPALL();
            }

            else
            {
                  printf("AntennaAutoTracker [options]\n");
                  printf("                    -t, track\n");
                  printf("                    -a, antenna position\n");
                  printf("                    -s, satellite
postion\n");
                  exit(1);
            }
      }
}

bool SatPosition(int *output_az, int *output_el)
{
      int err;
      FILE * pFile=NULL;
      long lSize;
      char * buffer;
      int azimuth, elevation;

      //call predict and output results to a file
```

```
	err=system("predict.exe -f OSCAR-45 > output.txt");  //should
have variables here
	if (err==-1){
		printf("[Error] executing predict\n");
		exit(1);
	}
	//else
		//printf("Predict successfully executed\n");

	pFile = fopen ( "output.txt" , "rb" );

	// obtain file size.
	fseek (pFile , 0 , SEEK_END);
	lSize = ftell (pFile);
	rewind (pFile);

	// allocate memory to contain the whole file.
	buffer = (char*) malloc (lSize);
	if (buffer == NULL){
		printf("[Error] nothing in file\n");
		exit (1);
	}

	// copy the file into the buffer.
	fread (buffer,1,lSize,pFile);

	//printf("lSize=%d\n%s\n",lSize,buffer);

	parsePredictLine(buffer, &azimuth, &elevation);

	//printf("\nAZ=%d deg\nEL=%d deg\n", azimuth, elevation);

	// terminate
	fclose (pFile);
	free (buffer);

	*output_az=azimuth;
	*output_el=elevation;


	if(azimuth>=0 && elevation>=0)
		return true;
	else
		return false;
}

void parsePredictLine(char * buffer, int *AZ, int *EL)
{
	char cAZ[4];
	char cEL[4];
	int i=0;

	for(i=0;i<4;i++)
	{
		cEL[i]=buffer[i+32];
		cAZ[i]=buffer[i+37];
	}
```

```
	*AZ=atoi(cAZ);
	*EL=atoi(cEL);
}

int  getNextPassAZ(void)
{
	int err;
	FILE * pFile=NULL;
	long lSize;
	char * buffer;
	int azimuth, elevation;

		//call predict and output results to a file
	err=system("predict.exe -p OSCAR-45 > output.txt");
		//should have variables here
	if (err==-1){
		printf("[Error] executing predict\n");
		exit(1);
	}
	//else
		//printf("Predict successfully executed\n");

	pFile = fopen ( "output.txt" , "rb" );

	// obtain file size.
	fseek (pFile , 0 , SEEK_END);
	lSize = ftell (pFile);
	rewind (pFile);

	// allocate memory to contain the whole file.
	buffer = (char*) malloc (lSize);
	if (buffer == NULL){
		printf("[Error] nothing in file\n");
		exit (1);
	}

	// copy the file into the buffer.
	fread (buffer,1,lSize,pFile);

	//printf("lSize=%d\n%s\n",lSize,buffer);

	parsePredictLine(buffer, &azimuth, &elevation);

	//printf("\nAZ=%d deg\nEL=%d deg\n", azimuth, elevation);

	// terminate
	fclose (pFile);
	free (buffer);

	return azimuth;

}

void readCurrentAntennaOrientation(float *AZ, float *EL)
{
	long errorcode;
```

```
	long idnum=-1;
	long demo=0;
	long stateIO=0;
	long numCh=4;
	long channels[4]={0,1,2,3};
	long gains[4]={0,0,0,0};
	long ov;
	float voltages[4]={0,0,0,0};


	errorcode = AISample
(&idnum,demo,&stateIO,0,1,numCh,channels,gains,0,&ov,voltages);

	//printf("\nAISample error = %d\n",errorcode);
	//printf("\nLocal ID = %d\n",idnum);
	//printf("AI0 = %f\n",voltages[0]);  //AZ channel
	//printf("AI1 = %f\n",voltages[1]);
	//printf("AI2 = %f\n",voltages[2]);  //EL channel
	//printf("AI3 = %f\n",voltages[3]);


	//system("cls");

	*EL = float( (EL_MAX_DEG - EL_MIN_DEG) / (EL_MAX_AD - EL_MIN_AD)
) * float(255.0/5.0) * voltages[2];

	*AZ = float( (AZ_MAX_DEG - AZ_MIN_DEG) / (AZ_MAX_AD - AZ_MIN_AD)
) * float(255.0/5.0) * voltages[0] -  float( (AZ_MAX_DEG - AZ_MIN_DEG)
/ (AZ_MAX_AD - AZ_MIN_AD) ) * float(AZ_MIN_AD);

	//printf("%f %f\n",*EL, *AZ);
}


void antennaMoveUp(void)
{
	long idnum=-1;
	long errorcode;

	errorcode = EDigitalOut(&idnum,0,EL_UP_CHANNEL,0,1);
		//channel 2 ON
	errorcode = EDigitalOut(&idnum,0,EL_DN_CHANNEL,0,0);
		// down channel off both shouldnt be on at same time
}

void antennaMoveDown(void)
{
	long idnum=-1;
	long errorcode;

	errorcode = EDigitalOut(&idnum,0,EL_DN_CHANNEL,0,1);
		//channel 3 ON
	errorcode = EDigitalOut(&idnum,0,EL_UP_CHANNEL,0,0);
}

void antennaRotatePos(void)
{
```

```
	long idnum=-1;
	long errorcode;

	errorcode = EDigitalOut(&idnum,0,AZ_POS_CHANNEL,0,1);  //channel
0 ON
	errorcode = EDigitalOut(&idnum,0,AZ_NEG_CHANNEL,0,0);
}

void antennaRotateNeg(void)
{
	long idnum=-1;
	long errorcode;

	errorcode = EDigitalOut(&idnum,0,AZ_NEG_CHANNEL,0,1);  //channel
1 ON
	errorcode = EDigitalOut(&idnum,0,AZ_POS_CHANNEL,0,0);
}

void antennaSTOPEL(void)
{
	long idnum=-1;
	long errorcode;

	errorcode = EDigitalOut(&idnum,0,EL_UP_CHANNEL,0,0);
	errorcode = EDigitalOut(&idnum,0,EL_DN_CHANNEL,0,0);
}

void antennaSTOPAZ(void)
{
	long idnum=-1;
	long errorcode;

	errorcode = EDigitalOut(&idnum,0,AZ_POS_CHANNEL,0,0);
	errorcode = EDigitalOut(&idnum,0,AZ_NEG_CHANNEL,0,0);
}


void antennaSTOPALL(void)
{
	long idnum=-1;
	long errorcode;

	errorcode = EDigitalOut(&idnum,0,EL_UP_CHANNEL,0,0);
	errorcode = EDigitalOut(&idnum,0,EL_DN_CHANNEL,0,0);
	errorcode = EDigitalOut(&idnum,0,AZ_POS_CHANNEL,0,0);
	errorcode = EDigitalOut(&idnum,0,AZ_NEG_CHANNEL,0,0);
}
```

Appendix F

Section F1: Terminal Node Controller (Server-Side)

```
/*
 * @author: Daniel Schuet
 * <A HREF="mailto:dschuet@scu.edu"> (Dan Schuet) </A>
 *
 * @Description: This class was change from its original design by
 * Sun Microsystems, and it has been configured to communicate
 * with a TNC/Modem attached to COM2. It is also designed to send &
 * receive data from RBNB DataTurbine.
 *
 * Any questions concerning this class, can be found online from
 * java.sun.com or rbnb.creare.com
 *
 * @(#)TNCDT.java 1.12 98/06/25 SMI
 *
 * Copyright (c) 1998 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license
 * to use, modify and redistribute this software in source and binary
 * code form, provided that i) this copyright notice and license appear
 * on all copies of the software; and ii) Licensee does not utilize the
 * software in a manner which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind.
 * ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES,
 * INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND
 * ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY
 * LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE
 * SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS
 * BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES,
 * HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING
 * OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control
 * of aircraft, air traffic, aircraft navigation or aircraft
 * communications; or in the design, construction, operation or
 * maintenance of any nuclear facility. Licensee represents and
 * warrants that it will not use or redistribute the Software for such
 * purposes.
 */

import java.io.*;
import java.util.*;
import javax.comm.*;
import com.rbnb.sapi.*;

public class TNCDT implements Runnable, SerialPortEventListener {
```

```
    static CommPortIdentifier portId;
    static Enumeration portList;
    static String messageString = "";

    static int stx = 0;
    static boolean outputBufferEmptyFlag = false;

    static InputStream  inputStream;
    static OutputStream outputStream;
    static SerialPort serialPort;
    Thread readThread;

    static Source source=new Source();
    static Sink sink    =new Sink();
    static ChannelMap sMap, rMap;

    static boolean dataReady=false;
    static String sBuffer="";
    static String nBuffer="";

    public static void main(String[] args) {
            String host;
            int i=0, marker=0;
            String star="*";

            System.out.println("Welcome to the TNC to Data Turbine
program!");
            if(args.length>0){
                  host=args[0];
            System.out.println("Connecting to " + host + "...");
            }
            else {
                  System.out.println("ERROR: No RBNB host defined");
            return;
            }
            try {
                  // Opens up an RBNB connection
                  source.OpenRBNBConnection(host, "satSource");
                  sink.OpenRBNBConnection(host,"satSink");

                  sMap = new ChannelMap();
                  sMap.Add("RXrawTNCstring");
                  sMap.PutTimeAuto("timeofday");
                  source.Register(sMap);

                  // Pull data from the server:
                  rMap = new ChannelMap();
                  rMap.Add("MatCmd/TXstring");

                  sink.Subscribe(rMap);

            } catch (SAPIException se) { se.printStackTrace(); }

            // stores all ports to an enumeration
            portList = CommPortIdentifier.getPortIdentifiers();

            // search through enumeration of ports for COM2
```

```
            while (portList.hasMoreElements()) {
                  portId = (CommPortIdentifier) portList.nextElement();
                  if (portId.getPortType() ==
CommPortIdentifier.PORT_SERIAL) {
                        if (portId.getName().equals("COM2")) {
                              TNCDT reader = new TNCDT();
                              System.out.println(portId.getName() + "
found!");
                        }
                  }
            }

            ChannelMap aMap;

            while(true){
                  //Do a bunch of pre-checks because everything does
                  //not end with a carriage return
                  if( sBuffer.endsWith("PRESS (*) TO SET BAUD RATE") )
                  {
                        try {
                              outputStream.write(star.getBytes());
                        } catch (IOException e) {}

                        System.out.println("Autobaud rate sequence
detected!");
                        sBuffer="";
                        marker=0;
                  }

                  if( sBuffer.endsWith("ENTER YOUR CALLSIGN=> ") )
                  {
                        try{
                              System.out.println("Placing string: " +
"ENTER YOUR CALLSIGN=>" + " into server.");
                              System.out.println(sBuffer.length());
                              sMap.PutDataAsString(0,"ENTER YOUR
CALLSIGN=>");
                              source.Flush(sMap);
                        } catch (SAPIException se) {
se.printStackTrace(); }

                        sBuffer="";
                        marker=0;
                  }

                  if( sBuffer.endsWith("sapphire>") )
                  {
                        try{
                                    System.out.println("Placing string:
" + "sapphire>" + " into server.");

      System.out.println(sBuffer.length());

      sMap.PutDataAsString(0,"sapphire>");
                              source.Flush(sMap);
                              } catch (SAPIException se) {
se.printStackTrace(); }
```

```
                        sBuffer="";
                        marker=0;
                  }

                  //Loop to check to see if to post to data turbine
(post off return carriage found)
                  for(i=marker; i<sBuffer.length(); i++)
                  {
                        if(sBuffer.charAt(i)=='\r')
                        {
                              nBuffer=sBuffer.substring(marker,i);
                              marker=i+1;
                              i=sBuffer.length();
                              System.out.println("marker=" + marker);

                              if (marker<=1)
                                    nBuffer="ok";

                              try{
                                    System.out.println("Placing string:
" + nBuffer + " into server.");

      System.out.println(sBuffer.length());
                                    sMap.PutDataAsString(0,new
String(nBuffer));
                                    source.Flush(sMap);

                              } catch (SAPIException se) {
se.printStackTrace(); }

                        }
                  }

                  try{
                        // do a non-blocking check to see if there is
                        // any information on DT
                        aMap = sink.Fetch(1, rMap);
                        if (aMap.NumberOfChannels()>0)
                        {
                              sBuffer="";
                  //clear and reset buffer so it doesn't get too big
                              marker=0;
                              System.out.println("Data ready to be
received from DT!");

                  //store information from DT to a String

      messageString=aMap.GetDataAsString(0)[0]+"\r";
                              System.out.println("Command from MATLAB:
" + messageString);
                              try {

      outputStream.write(messageString.getBytes());
                              } catch (IOException e) {}
                        }
                  } catch (SAPIException se) { se.printStackTrace(); }
            }
```

```
}

public TNCDT() {
    try {
        serialPort = (SerialPort) portId.open("TNCDTApp", 2000);
    } catch (PortInUseException e) {}
    try {
        inputStream = serialPort.getInputStream();
              outputStream = serialPort.getOutputStream();
    } catch (IOException e) {}
  try {
        serialPort.addEventListener(this);
        } catch (TooManyListenersException e) {}
    serialPort.notifyOnDataAvailable(true);
    try {
                    serialPort.setSerialPortParams(1200,
            SerialPort.DATABITS_8,
            SerialPort.STOPBITS_1,
            SerialPort.PARITY_NONE);
    } catch (UnsupportedCommOperationException e) {}
    readThread = new Thread(this);
    readThread.start();
}

public void run() {
    try {
        Thread.sleep(20000);
    } catch (InterruptedException e) {}
}

public void serialEvent(SerialPortEvent event) {
    switch(event.getEventType()) {
    case SerialPortEvent.BI:
    case SerialPortEvent.OE:
    case SerialPortEvent.FE:
    case SerialPortEvent.PE:
    case SerialPortEvent.CD:
    case SerialPortEvent.CTS:
    case SerialPortEvent.DSR:
    case SerialPortEvent.RI:
    case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
        break;
    case SerialPortEvent.DATA_AVAILABLE:
        byte[] readBuffer = new byte[40];
              int numBytes=0, j=0;

       try {
            //pull information from the Comm Port
            while (inputStream.available() > 0) {
                numBytes = inputStream.read(readBuffer);
            }
                    //Construct string
                    byte[] fBuffer = new byte[numBytes];
                    for(j=0; j<numBytes; j++)
                    {
                          fBuffer[j]=readBuffer[j];
                    }
```

```
                        sBuffer = sBuffer + new String(fBuffer);
                } catch (IOException e) {}
                break;
            }
        }
    }
```

Appendix G

Section G1: Transceiver (Client & Server Side)

```
/**
 *
 * The Transceiver class handles the validation, formatting, and
 * accessor methods for the transceiver commands.
 * Purpose: Designed for Santa University RACE 2003-2004 SeniorDesign
 * Team
 * @author
 *    Carleton Cheng
 *  <A HREF="mailto:C1Cheng@scu.edu"> (C1Cheng@scu.edu) </A>
 * @version
 *    Created: April 12, 2004
 *
 */

import java.io.*;
import java.util.*;
import java.lang.*;
import javax.swing.*;

public class Transceiver
{
      //variables and constants for Tranceiver
      private static final String preamble  = "FE";
      // fixed preamble
      private static final String dfaltXCRAddr = "60";
      // fixed default tranceiver address
      private static final String dfaltAddr = "00";
      // fixed default address
      private static String cmdNo = "00";
      // temporary command number
      private static String subCmdNo = "00";
      // temporary sub command number
      private static String dataBCD1,dataBCD2,dataBCD3,dataBCD4;
      // BCD portion of command (stored in an array)
      private static final String endOfMsg = "FD";
      // fixed end of message code
      private static final int BASE = 16;
      // base of number format for serial communication

      private int numberOfCommands;
      private String command[];

/*********************************************************************/
/*****************          Constructor        *********************/
/*********************************************************************/

      //Default Constructor
      public Transceiver()
      {
```

```
		dataBCD1 = "00";
		dataBCD2 = "00";
		dataBCD3 = "00";
		dataBCD4 = "00";

		numberOfCommands = 3;
		command = new String[numberOfCommands];
		command[0] = "xcr -sf"; //command sets frequency
		command[1] = "xcr -su"; //command selects uplink mode
		command[2] = "xcr -sd"; //command selects downlink mode
	}

/*********************************************************************/
/*****************          Accessors & Mutators          **********/
/*********************************************************************/


	//Accessor Methods
	public String getPreamble()
	{
		return preamble;
	}

	public String getDfaltXCRAddr()
	{
		return dfaltXCRAddr;
	}
	public String getDfaltAddr()
	{
		return dfaltAddr;
	}
	public String getCmdNo()
	{
		return cmdNo;
	}

	public String getSubCmdNo()
	{
		return subCmdNo;
	}

	public String getBCD(int value)
	{
		if (value == 1)
		{
			return dataBCD1;
		}
		if (value == 2)
		{
			return dataBCD2;
		}
		if (value == 3)
		{
			return dataBCD3;
		}
		if (value == 4)
		{
```

```
			return dataBCD4;
		}
		return null;
	}

	public String getEndOfMsg()
	{
		return endOfMsg;
	}

	public void setFrequency(String input)
	{
		StringTokenizer cmd = new
StringTokenizer(input.substring(8));
		char bcdValue[];

		bcdValue = input.substring(8).toCharArray(); //creating
array for values

		//MAKE SURE YOU CHECK IF THE COMMAND IS VALID

		//if format looks like xxx.xxx
		if(input.substring(8).length() == 7)
		{
			String temp1 = "0";
			String temp2 = ""+bcdValue[0];
			String temp3 = ""+bcdValue[1];
			String temp4 = ""+bcdValue[2];
			String temp5 = ""+bcdValue[4];
			String temp6 = ""+bcdValue[5];
			String temp7 = ""+bcdValue[6];
			String temp8 = "0";

			//BCD format, order must be reversed
			dataBCD1 = temp7 + temp8;
			dataBCD2 = temp5 + temp6;
			dataBCD3 = temp3 + temp4;
			dataBCD4 = temp1 + temp2;
		}
		//if format looks like xxx.xx
		else if(input.substring(8).length() == 6)
		{
			String temp1 = "0";
			String temp2 = ""+bcdValue[0];
			String temp3 = ""+bcdValue[1];
			String temp4 = ""+bcdValue[2];
			String temp5 = ""+bcdValue[4];
			String temp6 = ""+bcdValue[5];

			//BCD format, order must be reversed
			dataBCD2 = temp5 + temp6;
			dataBCD3 = temp3 + temp4;
			dataBCD4 = temp1 + temp2;
		}
		//if format looks like xxx.x
		else if(input.substring(8).length() == 5)
		{
```

```
            String temp1 = "0";
            String temp2 = ""+bcdValue[0];
            String temp3 = ""+bcdValue[1];
            String temp4 = ""+bcdValue[2];
            String temp5 = ""+bcdValue[4];
            String temp6 = "0";

            //BCD format, order must be reversed
            dataBCD2 = temp5 + temp6;
            dataBCD3 = temp3 + temp4;
            dataBCD4 = temp1 + temp2;
      }
      //if format looks like xxx
      else if(input.substring(8).length() == 3)
      {
            String temp1 = "0";
            String temp2 = ""+bcdValue[0];
            String temp3 = ""+bcdValue[1];
            String temp4 = ""+bcdValue[2];

            //BCD format, order must be reversed
            dataBCD3 = temp3 + temp4;
            dataBCD4 = temp1 + temp2;
      }
      //if format looks like xxx.xxx.x
      else
      {
            String temp1 = "0";
            String temp2 = ""+bcdValue[0];
            String temp3 = ""+bcdValue[1];
            String temp4 = ""+bcdValue[2];
            String temp5 = ""+bcdValue[4];
            String temp6 = ""+bcdValue[5];
            String temp7 = ""+bcdValue[6];
            String temp8 = ""+bcdValue[8];

            //BCD format, order must be reversed
            dataBCD1 = temp7 + temp8;
            dataBCD2 = temp5 + temp6;
            dataBCD3 = temp3 + temp4;
            dataBCD4 = temp1 + temp2;
      }
}

public void setULink()
{
      cmdNo = "07";
      subCmdNo = "D0";
}

public void setDLink()
{
      cmdNo = "07";
      subCmdNo = "D1";
}

public void resetToDefault()
```

```
	{
		cmdNo = "00";
		subCmdNo = "00";
		dataBCD1 = "00";
		dataBCD2 = "00";
		dataBCD3 = "00";
		dataBCD4 = "00";
	}

	//function: validates the XCR Command
	//---------------------------------------------------------------
	//		THIS FUNCTION VALIDATES THE MAIN COMMAND
	//		RETURNS: A BOOLEAN
	//---------------------------------------------------------------
	public boolean validCommand(String input)
	{
		for(int i=0; i<numberOfCommands; i++)
		{
			if (input.toLowerCase().startsWith(command[i]))
			{
				if(input.toLowerCase().startsWith(command[0]))
				{
					return this.validFrequency(input);
				}
				else
					return true;
			}
		}
		return false;
	}

	//---------------------------------------------------------------
	//		THIS FUNCTION VALIDATES THE FREQUENCY
	//		RETURNS: A BOOLEAN
	//---------------------------------------------------------------
	public boolean validFrequency(String input)
	{
		try
		{
			StringTokenizer cmd = new
StringTokenizer(input.substring(8));
			for(int i=0; cmd.hasMoreElements();i++)
			{
				int freqVal =
Integer.parseInt(cmd.nextToken("."),10);
				if((0 <= freqVal) || (freqVal < 9999999))
				{
					return true;
				}
			}
		}
		catch(Exception e) {}
			return false;
	}
}
```

Section G2: Transceiver Interpreter (Client-Side)

```
/**
 *
 * The TransceiverInterpreter class handles validating a user command
 * and sending the command to the transceiver hardware.
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * @author
 *    Carleton Cheng
 *  <A HREF="mailto:C1Cheng@scu.edu"> (C1Cheng@scu.edu) </A>
 * @version
 *    Created: April 12, 2004
 *
 */

import java.io.*;
import java.util.*;
import java.lang.*;
import javax.swing.*;

public class TransceiverInterpreter
{
      private String host;
      private static DataTurbine dt = new DataTurbine();
      private static DataTurbine dt2 = new DataTurbine();
      private int numberOfCommands;
      private String command[];

/*******************************************************************/
/******************          Constructor       ********************/
/*******************************************************************/

      public TransceiverInterpreter()
      {
            numberOfCommands = 3;
            command = new String[numberOfCommands];
            command[0] = "xcr -sf"; //command sets frequency
            command[1] = "xcr -su"; //command selects uplink mode
            command[2] = "xcr -sd"; //command selects downlink mode
      }

      public TransceiverInterpreter(String host)
      {
            numberOfCommands = 4;
            command = new String[numberOfCommands];
            command[0] = "xcr -sf"; //command sets frequency
            command[1] = "xcr -su"; //command selects uplink mode
            command[2] = "xcr -sd"; //command selects downlink mode
            command[3] = "xcr -help"; //command selects help menu

            this.host = host;
            dt.openSinkConnection(host,"TransceiverSink");
            dt.addChannelSend("/sattest/sendTransceiver/text");
            dt2.openSinkConnection(host,"TransceiverSink");
            dt2.addChannelSend("/sattest/receiveTransceiver/text");
```

```
	}

/*****************************************************************/
/*****************           Accessors & Mutators         **********/
/*****************************************************************/

	public boolean validateCommand(String input)
	{	/**
		 *
		 * This function checks whether or not a command is valid
		 * or not.
		 * @param input
		 *   the command in the following format: XCR [-subcommand]
		 *
		 *          +functions: -sf <frequency value>   sets the
		 *          frequency of the following form:
		 *
		 *          xxx.xxx.x
		 *
		 *          xxx.xxx
		 *
		 *          xxx
		 *          -su   sets Transceiver to UPLINK mode
		 *          -sd   sets Transceiver to DOWNLINK mode
		 * @return
		 *   boolean
		 */
		for(int i=0; i<numberOfCommands; i++)
		{
			if (input.toLowerCase().startsWith(command[i]))
			{
				if(input.toLowerCase().startsWith(command[0]))
				{
					return validFrequency(input);
				}
				else
					return true;
			}
		}
		return false;
	}

	private boolean validFrequency(String input)
	{
		/**
		 *
		 * This function checks whether or not a frequency is valid
		 * or not.
		 *
		 * @param input
		 *   the entire command in the following format: XCR [-
		 * subcommand]
		 *
		 *          + functions: -sf <frequency value>  sets the
		 *          frequency of the following form:
		 *
		 *          xxx.xxx.x <x = 0-9>
```

```
             *
             *          xxx.xxx       <x = 0-9>
             *
             *          xxx           <x = 0-9>
             *          -su   sets Transceiver to UPLINK mode
             *          -sd   sets Transceiver to DOWNLINK mode
             * @return
             *   boolean
             *
             */
            try
            {
                  StringTokenizer cmd = new
StringTokenizer(input.substring(8));
                  for(int i=0; cmd.hasMoreElements();i++)
                  {
                        int freqVal =
Integer.parseInt(cmd.nextToken("."),10);
                        if((0 <= freqVal) || (freqVal <= 9999999))
                        {
                              return true;
                        }
                  }
            }
            catch(Exception e) {}
                  return false;
      }

      public void sendCommand(String cmd,int wait)
      {
            /**
             *
             * This function checks then sends the command to the
             * transceiver hardware.
             * @param input
             *   the command in the following format: XCR [-subcommand]
             *
             *          +functions: -sf <frequency value>   sets the
             *          frequency of the following form:
             *
             *          xxx.xxx.x
             *
             *          xxx.xxx
             *
             *          xxx
             *          -su   sets Transceiver to UPLINK mode
             *          -sd   sets Transceiver to DOWNLINK mode
             * @return
             *   void
             */
                  if ( cmd != null )
                  {
                        if (validateCommand(cmd) == true)
                        {
                              //System.out.println("This is what you
just typed: " + cmd + "\n");
                              dt.sendMessage(cmd,wait);
```

```
                              if(cmd.startsWith(command[0]))
                              {
                                    dt2.sendMessage("Transceiver 
Frequency Set.",wait);
                              }
                              if(cmd.startsWith(command[1]))
                              {
                                    dt2.sendMessage("Transceiver Uplink 
Mode Set.",wait);
                              }
                              if(cmd.startsWith(command[2]))
                              {
                                    dt2.sendMessage("Transceiver 
Downlink Mode Set.",wait);
                              }
                              if(cmd.startsWith(command[3]))
                              {
                                    dt2.sendMessage(getHelp(),wait);
                              }
                        }
                  }
      }

      public static String getHelp()
      {
            String help1 = "------------------------\nHelp File for 
Transceiver\n------------------------\n\n";
            String help2 = "commands:\n   -sf <frequency>\tsets the 
frequency to the given <frequency>\n\n";
            String help3 = "   -su\t\t\tsets the mode to uplink\n\n";
            String help4 = "   -sd\t\t\tsets the mode to downlink\n\n
-help";

            String help = help1 + help2 + help3 + help4;

            return help;
      }
}
```

Section G3: Transceiver Echo (Client-Side)

```
/**
 * @Author: Carleton Cheng
 *   <A HREF="mailto: C1Cheng@scu.edu"> (C1Cheng@scu.edu) </A>
 *
 * <P>
 * @version
 * Created: 2/18/2004
 * <P>
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 * <P>
 * Description: This class is used for receiving all information coming
 * from the Transceiver located at the Ground Station.  This class uses
 * RBNB Data Turbine to establish a connection with the ground station
 * located at Santa Clara University Engineering Department. For more
 * information on DataTurbine and it's functionality, please refer to
 * the RBNB website: http://rbnb.creare.com
 **/

import java.io.*;
import java.util.*;
import java.lang.*;

public class TransceiverClientEcho
{
      public static void main(String[] args)
      {
            DataTurbine dt = new DataTurbine();
                  //DataTurbine Class used for connection
            String host;
                  //Variable used to specify where DT Host is

            System.out.println("RACE: Transceiver Response Window\n");
            if(args.length>0)
            {
                  host=args[0];
                        //user must specify where DT Host is
                  System.out.println("Connecting to " + host + "...");
            }
            else
            {
                  System.out.println("ERROR: No RBNB host defined");
                  return;
            }

//Establish connection with Ground Station by setting up channels
            dt.openPluginConnection(host, "receiveTransceiver");

            dt.addChannelReceive("text");
            dt.registerReceiveChannel();

            System.out.println("Connected...");

            while (true)
```

```
		{
			String message = dt.receiveMessage(-1);
				//blocking wait for information

			if (message != null) {
				System.out.println(message);
					//if there's a message, output it
				System.out.println();
			}
		}
	}
}
```

Section G4: Transceiver Server (Server-Side)

```
/*
 * @Author: Carleton Cheng
 *    <A HREF="mailto:C1Cheng@scu.edu"> (C1Cheng@scu.edu) </A>
 *
 * @version
 * Created: 2/23/2004
 *
 * Purpose: Designed for Santa University RACE 2003-2004 Senior Design
 * Team
 *
 * Description: This class takes request coming in through DT and
 * forwards the command out to the Transceiver that is connected to the
 * Serial Port. any response coming from the Transceiver is forwarded
 * back out through RBNB Data Turbine.
 *
 * @(#)TNCDT.java 1.12 98/06/25 SMI
 *
 * Copyright (c) 1998 Sun Microsystems, Inc. All Rights Reserved.
 *
 * Sun grants you ("Licensee") a non-exclusive, royalty free, license
 * to use, modify and redistribute this software in source and binary
 * code form, provided that i) this copyright notice and license appear
 * on all copies of the software; and ii) Licensee does not utilize the
 * software in a manner which is disparaging to Sun.
 *
 * This software is provided "AS IS," without a warranty of any kind.
 * ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES,
 * INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A
 * PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND
 * ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY
 * LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE
 * SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS
 * BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT,
 * INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES,
 * HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING
 * OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 *
 * This software is not designed or intended for use in on-line control
 * of aircraft, air traffic, aircraft navigation or aircraft
 * communications; or in the design, construction, operation or
 * maintenance of any nuclear facility. Licensee represents and
 * warrants that it will not use or redistribute the Software for such
 * purposes.
 */

import java.io.*;
import java.util.*;
import javax.comm.*;

public class TransceiverServer implements Runnable,
SerialPortEventListener {
      static CommPortIdentifier portId;
      static Enumeration portList;
```

```
    static String messageString = "";

    static int stx = 0;
    static boolean outputBufferEmptyFlag = false;

    static InputStream      inputStream;
    static OutputStream     outputStream;
    static SerialPort serialPort;
    Thread readThread;

    static boolean portfound = false;
    static boolean dataReady = false;
    static String sBuffer = "";
    static int num;
    private static final int BASE = 16;
    private int numberOfCommands;
    private static String command[];

  public static void main(String[] args) {
          DataTurbine transceiverDT = new DataTurbine();
          String host;      //variable to store address of host
          String plugInName = "sendTransceiver";
                      // name of the plugin that will be used
                      // to read information from DT.
          String sendChannel = "/sattest/receiveTransceiver/text";
                      // the location of the channel
                      // to write information to DT.
          String commport = "COM1";
                      // the desired serial port to communicate on

          Transceiver handleXCR = new Transceiver();

          System.out.println("Welcome to the Transceiver to Data
Turbine program!");
          if(args.length>0){
                host=args[0];
          System.out.println("Connecting to " + host + "...");
          }
          else {
                System.out.println("ERROR: No RBNB host defined");
          return;
          }

    // plugin used to create a channel that you will read from when
    // client writes to channel
          transceiverDT.openPluginConnection(host,plugInName);
          transceiverDT.addChannelReceive("text");
          transceiverDT.registerReceiveChannel();

    // sink used to write a channel

    transceiverDT.openSinkConnection(host,"TransceiverServerSink");
          transceiverDT.addChannelSend(sendChannel);

          System.out.println("Connected...");

                      //Search for the COM1
```

```
            portList = CommPortIdentifier.getPortIdentifiers();

            while (portList.hasMoreElements()) {
                        portId = (CommPortIdentifier)
portList.nextElement();
                        if (portId.getPortType() ==
CommPortIdentifier.PORT_SERIAL) {
                        if (portId.getName().equals(commport)) {
                                    TransceiverServer transceiver = new
TransceiverServer();
                              System.out.println(portId.getName() + "
found!");
                              portfound = true;
                        }
                        }
            }

            if (portfound == false)
            {
                  System.out.println("Port was not found...\n\nExiting
program....");
            }

            while(portfound){
                  if(dataReady){
                        try {
                        Thread.sleep(200);//ms wait to finish sending
data
                        // Push data onto the server:

                        //check if there is something in the buffer
                        if (sBuffer != null & sBuffer != "")
                        {
                              System.out.println("Placing string: " +
sBuffer + " into server.");
                              transceiverDT.sendMessage(sBuffer,1);
                        }

                              //reset flags
                        dataReady=false;
                        sBuffer="";

                        } catch (InterruptedException e) {}
                  }

                  messageString = transceiverDT.receiveMessage(1);

                  if (messageString != null)
                  {
                        stx=1;//clear buffer
                        System.out.println("Data ready to be received
from DT!");

                        System.out.print("Command: " + messageString +
" ");

                        //---------------NEW-------------------------//
```

```
                        if
(handleXCR.validCommand(messageString)==true)
                        {
                              try {
                                    outputStream =
serialPort.getOutputStream();
                               } catch (IOException e) {}

                        try {
                              Thread.sleep(1000);  // Be sure data is
xferred before closing
                        } catch (Exception e) {}


      if(messageString.toLowerCase().startsWith(command[0]))
                              {
                                    try
                                    {

      handleXCR.setFrequency(messageString);

      outputStream.write(Integer.parseInt(handleXCR.getPreamble(),BASE)
);

      outputStream.write(Integer.parseInt(handleXCR.getPreamble(),BASE)
);

      outputStream.write(Integer.parseInt(handleXCR.getDfaltXCRAddr(),B
ASE));

      outputStream.write(Integer.parseInt(handleXCR.getDfaltAddr(),BASE
));

      outputStream.write(Integer.parseInt(handleXCR.getCmdNo(),BASE));

      outputStream.write(Integer.parseInt(handleXCR.getSubCmdNo(),BASE)
);

      outputStream.write(Integer.parseInt(handleXCR.getBCD(1),BASE));

      outputStream.write(Integer.parseInt(handleXCR.getBCD(2),BASE));

      outputStream.write(Integer.parseInt(handleXCR.getBCD(3),BASE));

      outputStream.write(Integer.parseInt(handleXCR.getBCD(4),BASE));

      outputStream.write(Integer.parseInt(handleXCR.getEndOfMsg(),BASE)
);
                                          handleXCR.resetToDefault();
                                    }
                                    catch (IOException e) {}
                              }

      if(messageString.toLowerCase().startsWith(command[1]))
                              {
                                    try
                                    {
```

```
                                        handleXCR.setULink();


      outputStream.write(Integer.parseInt(handleXCR.getPreamble(),BASE)
);

      outputStream.write(Integer.parseInt(handleXCR.getPreamble(),BASE)
);

      outputStream.write(Integer.parseInt(handleXCR.getDfaltXCRAddr(),B
ASE));

      outputStream.write(Integer.parseInt(handleXCR.getDfaltAddr(),BASE
));

      outputStream.write(Integer.parseInt(handleXCR.getCmdNo(),BASE));

      outputStream.write(Integer.parseInt(handleXCR.getSubCmdNo(),BASE)
);

      outputStream.write(Integer.parseInt(handleXCR.getEndOfMsg(),BASE)
);
                                        handleXCR.resetToDefault();
                                  }
                                  catch (IOException e) {}
                            }

      if(messageString.toLowerCase().startsWith(command[2]))
                            {
                                  try
                                  {
                                        handleXCR.setDLink();


      outputStream.write(Integer.parseInt(handleXCR.getPreamble(),BASE)
);

      outputStream.write(Integer.parseInt(handleXCR.getPreamble(),BASE)
);

      outputStream.write(Integer.parseInt(handleXCR.getDfaltXCRAddr(),B
ASE));

      outputStream.write(Integer.parseInt(handleXCR.getDfaltAddr(),BASE
));

      outputStream.write(Integer.parseInt(handleXCR.getCmdNo(),BASE));

      outputStream.write(Integer.parseInt(handleXCR.getSubCmdNo(),BASE)
);

      outputStream.write(Integer.parseInt(handleXCR.getEndOfMsg(),BASE)
);
                                        handleXCR.resetToDefault();
                                  }
                                  catch (IOException e) {}
                            }
```

```
                        }

                        System.out.println(" sent to serial...\n");

                        try {
                              Thread.sleep(200);
                        } catch (Exception e) {}

                        // sBuffer = "some string value to write to
Echo";
                        // dataReady = true;
                        stx = 0;
                  }
            }
    }

    public TransceiverServer() {

            numberOfCommands = 3;
            command = new String[numberOfCommands];
            command[0] = "xcr -sf"; //command sets frequency
            command[1] = "xcr -su"; //command selects uplink mode
            command[2] = "xcr -sd"; //command selects downlink mode

        try {
            serialPort = (SerialPort) portId.open("TNCDTApp", 2000);
        } catch (PortInUseException e) {}
        try {
            inputStream = serialPort.getInputStream();
                  outputStream = serialPort.getOutputStream();
        } catch (IOException e) {}
      try {
            serialPort.addEventListener(this);
            } catch (TooManyListenersException e) {}
        serialPort.notifyOnDataAvailable(true);
        try {
                        serialPort.setSerialPortParams(9600,
                SerialPort.DATABITS_8,
                SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE);
        } catch (UnsupportedCommOperationException e) {}
        readThread = new Thread(this);
        readThread.start();
    }

    public void run() {
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {}
    }

    public void serialEvent(SerialPortEvent event) {
        switch(event.getEventType()) {
        case SerialPortEvent.BI:
        case SerialPortEvent.OE:
        case SerialPortEvent.FE:
        case SerialPortEvent.PE:
```

```
        case SerialPortEvent.CD:
        case SerialPortEvent.CTS:
        case SerialPortEvent.DSR:
        case SerialPortEvent.RI:
        case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
            break;
        case SerialPortEvent.DATA_AVAILABLE:
            byte[] readBuffer = new byte[24];

           try {
                while (inputStream.available() > 0) {
                    int numBytes = inputStream.read(readBuffer);
                }
                        if(stx==0){
                              dataReady=true;
                              sBuffer = sBuffer + new
String(readBuffer);
                        }

            } catch (IOException e) {}
            break;
        }
    }
}
```

Appendix H

Section H1: Server Side Directory Structure

A series of batch files were created to make it easier to start up the Server Side Applications and Services. The Server Side directory, when unzipped, is formated as shown below. There is a specific batch file that loads up a specific device/service in each folder. More information about each individual batch file can be found below in Section 2.

There is a master batch file that loads up all the services and device specific batch files in the folders. This batch file is located in the root folder of dtHardware. More information about the main batch file can be found in Section 3.

Section H2: Individual Batch Files

<dtHardware>\Antenna_And_Predict\runAntennaServer.bat

```
java AntennaServer localhost:3333

@pause
```

<dtHardware>\Antenna_And_Predict\runPredictServer.bat

```
rem The first two commands are necessary
rem There was an error when they were not
rem invoked before invoking the PredictServer class

predict -f OSCAR-45

predict -p OSCAR-45

java PredictServer localhost:3333

@pause
```

<dtHardware>\PowerStrip\runPowerServer.bat

```
java PowerStripServer localhost:3333

@pause
```

<dtHardware>\RBNB\runRBNB_No_Security.bat

```
rem This assumes you installed RBNB into
```

```
rem C:\Program Files
rem Also note that this is RBNB_V2.1

cd C:\Program Files\RBNB_V2.1\bin

rem this invokes the RBNB server on the localhost
rem and names the server "sattest"
java –jar rbnb.jar –a localhost:3333 –n sattest
```

<dtHardware>\TNC\runTNCServer.bat

```
java TNCDT localhost:3333

@pause
```

<dtHardware>\XCR\runXCRServer.bat

```
java TransceiverServer localhost:3333

@pause
```

Section H3: Main Batch File

<dtHardware>\RACE_Server.bat

```
@echo off

Echo Start RBNB Server

pushd RBNB
start runRBNBServer_No_Security
popd
rem pause

@pause

Echo Start Applications

pushd Antenna_And_Predict
start runAntennaServer
popd
rem pause

pushd Antenna_And_Predict
start runPredictServer
popd
rem pause

pushd PowerStrip
start runPowerServer
popd
rem pause

pushd TNC
start runTNCServer
popd
rem pause

pushd XCR
```

```
start runXCRServer
popd
rem pause

echo Finished starting RBNB Server and Applications
```

Section H4: Installation Instructions

```
-------------
INSTALLATION:
-------------

I.  System Requirements
    * 233 MHz Processor (recommend 800+ MHz)
    * 128 MB RAM (recommend 256+ MB RAM)
    * Internet Connection (preferable broadband connection)
    * installed or able to install Java Runtime Environment

    NOTE: System has been tested to run under Windows2000 and XP.


II. Setup
    1. Go to http://rbnb.creare.com
      a. click on downloads on the side bar
      b. download "Windows self-installing executable"
            - If you don't have the Java Virtual machine installed on
your machine
              choose "Windows (includes JVM)"
      c. after download, open file and follow installation instructions

    2. Go to java.sun.com
      a. Click on "Downloads" on the sidebar
      b. download the latest version of J2SE
            - or at least J2SE v1.4.2 w/Java Runtime Environment(JRE)
      c. install to desired location

    3. Set environment variables
      a. set classpath (under "user variables") to:
            <dir of DataTurbine>\RBNB_V2.1\bin\rbnb.jar;
            <dir of DataTurbine>\RBNB_V2.1\bin\source.jar;
            <dir of DataTurbine>\RBNB_V2.1\bin\rbnbjview.jar;
            <dir of DataTurbine>\RBNB_V2.1\bin\rbnbjcap.jar; .  *

                  * (make sure "." is at the end of the classpath as
                     shown above)

      b. set Path (under "system variables") to:
            <dir of jdk>\bin\; **

                  ** (make sure to set this path in front of other
                      system paths)

    4. Connect & Install devices to the appropriate port
      a. Labjack* (Antenna Controller) connected to both the Antenna
         Controller** the computer through any USB port.
      b. Serial Power Strip*** connected to "COM3" (Serial Port 3).
      c. TNC (Packet Modem)**** connected to "COM2" (Serial Port 2).
      d. Transceiver***** connected to "COM1" (Serial Port 1).

                  *      Labjack can found and purchased online at
                         http://www.labjack.com/
                  **     The Antenna Controller that is located at the
```

```
              Santa Clara University Ground Station is the
              "YAESU G-5500".
       ***    The Serial Power Strip that is located at the
              Santa Clara University Ground Station is the
              "Baytech RPC2".
              More information can be found at
              http://www.kvms.com/baytech/baytech_rpc2.asp
       ****   The TNC that is located at Santa Clara
              University is the "Kantronics 9612 Packet
              Creator".
       *****  The Transceiver that is located at Santa Clara
              University is the "ICOM 910 Dual Band
              Transceiver".

 NOTE: For further information on the RACE System and it's
       components, please refer to the RACE 2001-2002 and
       RACE 2003-2004 Senior Design Reports.


5. Download RACE files (zipped file)
  a. Extract contents into desired folder
  b. run batch file located in the Root RACE directory to start
     RACE Server
        <dir of RACE>\dtHardware\RACE_Server.bat
     See "Instructions On Using RACE Server" for more information
     on starting, stopping, and handling errors on the RACE Server
     Applications.

  NOTE: To run the batch file from your desktop, you can create a
        shortcut to these batch files.
```

Section H5: Using RACE Server Program

```
--------------------------------------
Instructions On How to Use RACE Server
--------------------------------------

After Unzipping the Race Server Program, you should see one batch file
in the root directory, <dir of dtHardware>\RACE_Server.bat

1. Start up Tomcat

2. To start the RBNB Server and Applications, double-click
   RACE_Server.bat

3. This brings up two windows
      a. One window will start up the RBNB Server.  Wait for the words
         "Started at address..." shows on the screen.
      b. The other window will say "Start RBNB Server" and then prompt
         you to "Press any key to continue".  ONLY continue once the
         first window says "Started at address..."
```

NOTE: An error will occur with the Server application programs if you do not wait for the RBNB Server to start.

4. Select the second window that prompts you to press any key

5. Press any key

6. Five more dos-command prompt windows will pop up. Each one has a Welcome Screen, and say that it is "connecting..." to the RBNB Server.
   a. A Successful connection will print out "connected..."
      --> This means that you can now connect to the ground station using the client side application.
   b. An Unsuccessful connection will print out Several RBNB Errors

----
Error Handling

If an error occurs:

1. close and shut down all dos-command windows that were opened.
2. check to see all devices were installed and connected to the correct port <See Installation Instructions for details>
3. Make sure RBNB was installed correctly
4. Make sure Tomcat was started successfully
5. Make sure RBNB Server was started successfully

If there are any further questions, please contact:

Peter Salas ... psalas@scu.edu
Carleton Cheng ... c1cheng@scu.edu

Appendix I

Section I1: Client Side Directory Structure

A series of batch files were created to make it easier to start up the Client Side Applications. The Client Side directory, when unzipped, is formated as shown below. There is a specific batch file that loads up a specific device in the Class Files folder. More information about each individual batch file can be found below in Section 2.

There is a master batch file that loads up all the device specific batch files in the Class Files folder. This batch file is located in the root folder of RACE Client Programs. More information about the main batch file can be found in Section 3.

Section I2: Individual Batch Files

<RACE Client Programs>\Class Files\RACE_Antenna_Echo.bat

```
@echo off
java AntennaClientEcho 129.210.19.118:3333

@pause
```

<RACE Client Programs>\Class Files\RACE_Client_Input.bat

```
@echo off
java ClientInput 129.210.19.118:3333
```

<RACE Client Programs>\Class Files\RACE_Power_Echo.bat

```
@echo off
java PowerStripClientEcho 129.210.19.118:3333

@pause
```

<RACE Client Programs>\Class Files\RACE_Predict_Echo.bat

```
@echo off
java PredictClientEcho 129.210.19.118:3333

@pause
```

<RACE Client Programs>\Class Files\RACE_Transceiver_Echo.bat

```
@echo off
java TransceiverClientEcho 129.210.19.118:3333

@pause
```

Section I3: Main Batch File

<RACE Client Programs>\RACE_Program.bat

```
@echo off

echo Invoking all Client Side Programs

pushd Class Files
start RACE_Client_Input
popd
rem pause

pushd Class Files
start RACE_Power_Echo
popd
rem pause

pushd Class Files
start RACE_Predict_Echo
popd
rem pause

pushd Class Files
start RACE_Antenna_Echo
popd
rem pause

pushd Class Files
start RACE_Transceiver_Echo
popd
rem pause

echo Finished
```

Section I4: Picture of RACE Client Program

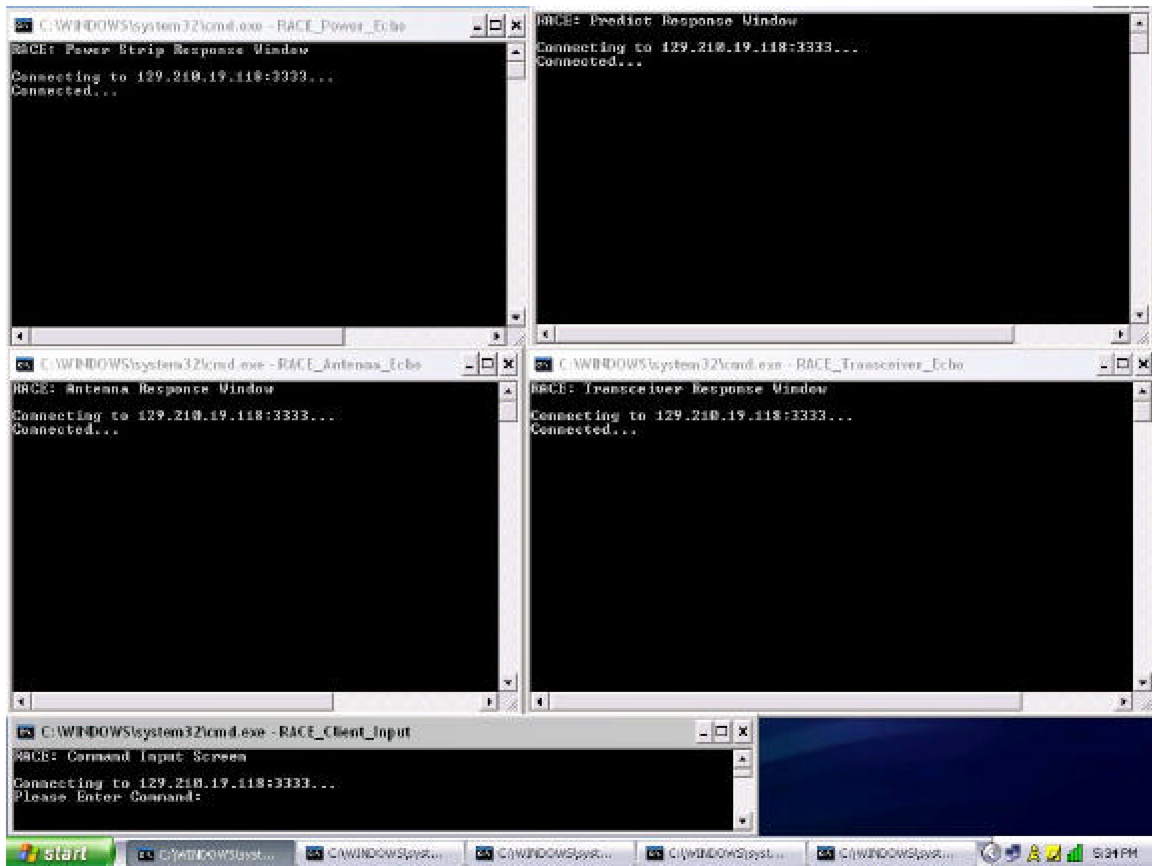

Figure I-1: RACE Client Program

Section I5: Installation Instructions

```
-------------
INSTALLATION:
-------------

I.  System Requirements
    * 233 MHz Processor
    * Internet Connection (preferable broadband connection)
    * installed or able to install Java Runtime Environment

    NOTE: System has been tested to run under Windows2000 and XP.
          Although it has yet to be tested under


II. Setup
    1. Go to http://rbnb.creare.com
      a. click on downloads on the side bar
      b. download "Windows self-installing executable"
            - if you don't have the Java Virtual machine installed on
              your machine choose "Windows (includes JVM)"
      c. after download, open file and follow installation instructions

    2. Go to java.sun.com
      a. Click on "Downloads" on the sidebar
      b. download the latest version of J2SE
            - or at least J2SE v1.4.2 w/Java Runtime Environment(JRE)
      c. install to desired location

    3. Set environment variables
      a. set classpath (under "user variables") to:
            <dir of DataTurbine>\RBNB_V2.1\bin\rbnb.jar;
            <dir of DataTurbine>\RBNB_V2.1\bin\source.jar;
            <dir of DataTurbine>\RBNB_V2.1\bin\rbnbjview.jar;
            <dir of DataTurbine>\RBNB_V2.1\bin\rbnbjcap.jar; .  *

                  * (make sure "." is at the end of the classpath as
                     shown above)

      b. set Path (under "system variables") to:
            <dir of jdk>\bin\; **

                  ** (make sure to set this path in front of other
                      system paths)

    4. Download RACE files
      a. Extract into desired folder
      b. run batch file located in the Root RACE directory

      NOTE: To run the batch file from your desktop, you can create a
            shortcut to this batch file.
```

Section I6: Using RACE Client Program

```
-------------------------------
Instructions On How to Use RACE
-------------------------------

You should see 1 batch file upon opening the folder (The root folder of
RACE).  Once clicked, you will see 5 other windows pop up onto your
screen.  Below, you will find descriptions of each window that pops up,
and how they work.


1) RACE_Client_Input
--------------------

   Description: This batch file opens up the command prompt window for
                sending commands to the ground station located
                at Santa Clara University.  Upon opening the
                RACE_Client_Input, you will see the words
                "Connecting to 129.210.19.118:3333".   At this point,
                the program is attempting to establish a connection
                with the Server.  If successfully connected, the words
                "Please Enter Command: " appear.  Else, you will see
                a DataTurbine Error.  Here are a list of commands
                available:

   NOTE: If you get a DataTurbine Error, the system will NOT work.
         Contact the system administrator if this happens.

   Commands:                    Description:
-----------------------------------------------------------------------
*  predict -help                Displays the help menu for controlling
                                Predict, which is the satellite
                                prediction software.  It is similar to
                                the NOVA software, and all commands that
                                deal with gathering information about
                                current possition or future pass
                                of a specific satellite, should be
                                directed here.

*  predict -update <Keplerian>      Updates the Keplerian elements

*  predict -poss <Satellite>    Requests the current possition of the
                                Satellite.

*  predict -pass <Satellite>    Requests when the next time the Satellite
                                will be in view.

*  power -help                  controls power of (1) Level-Converter for
                                Transceiver, (2) Transceiver,
                                (3) TNC/Modem, (4) and the Antenna
                                Controller.  This displays all the
                                commands necessary in controlling this
                                device.

*  power -on                    turns on all devices
```

*	power -off	turns off all devices
*	power -on lvconvert	turns on the level converter. This is necessary in sending commands to the Transceiver.
*	power -off lvconvert	turns off the level converter.
*	power -on xcr	turns on the Transceiver.
*	power -off xcr	turns off the Transceiver.
*	power -on modem	turns on the TNC/modem.
*	power -off modem	turns off the TNC/modem.
*	power -on antenna	turns on the Antenna Controller.
*	power -off antenna	turns off the Antenna Controller.
*	xcr -help	Displays the commands necessary in controlling the Transceiver.
*	xcr -sf <frequency>	sets the frequency to the given <frequency>. The format of the frequency should be of either of the three formats: (1) xxx (2) xxx.xxx (3) xxx.xxx.x
*	xcr -su	sets to the uplink mode. In this mode, you can perform the set frequency command (see above) and set the uplink frequency.
*	xcr -sd	sets to the downlink mode. In this mode, you can perform the set frequency command (see above) and set the downlink frequency.
*	<Invalid Command>	Displays an error message directing how one should type in a command.
*	antenna -help	Displays the commands necessary in controlling the Antenna(labjack)
*	antenna -t	Turns Autotracking on. If SAPPHIRE is not in view, then it will position the antenna to the next pass of the satellite.
*	antenna -a	Gives the current position of the antenna.
*	antenna -s	Gives the current position of SAPPHIRE (satellite).

```
2) RACE_Power_Echo
------------------

   Description: Controls all information returning from the device.
                This Echo's back anything received from the
                Serial Power Strip.



3) RACE_Predict_Echo
--------------------

   Description: Controls all information returning from the device.
                This Echo's back anything received from the
                Predict Software located on the Server.



4) RACE_Transceiver_Echo
------------------------

   Description: Controls all information returning from the device.
                This Echo's back anything received from the
                Transceiver.  If the server successfully performs a
                command, a success message is displayed here.

5) RACE_Antenna_Echo
--------------------

   Description: Controls all information returning from the device.
                This Echo's back anything received from the
                Labjack device located on the Server.
```

Section I7: Matlab Configuration and User Manual

Daniel Schuet, 2004

What you'll need:

1. Current version of Matlab (version 6.5 tested to work). Need to make sure that it has integrated java and the timer function.
2. RBNB Data Turbine v2.1. Installation instructions follow below.
3. Current version of SapphireTerm written by Daniel Schuet. These are Matlab m-files that allow you to hop onto the DataTurbine and control the ground station packet modem, along with some neat added features.
4. M_Map - a mapping package for Matlab [optional]. It can be found here: http://www2.ocgy.ubc.ca/~rich/map.html Follow the instructions in the user guide to install (basically its just a setting a path in Matlab).
5. Predict, AtomTime, http_get bundled zip file [optional]. Adds features of satellite tracking and predicting passes in Matlab. Extract to C:\

In order to connect to the ground station through the Internet some software will need to be set up and installed on the client side computer. The RBNB (Ring Buffered Network Bus) Data Turbine is used to gain connectivity to the capabilities of the ground station, i.e. the packet modem, the transceiver, and antenna auto tracking.

- Download and install RBNB Data Turbine v2.1. It can be found here: http://outlet.creare.com/rbnb/download.V2.1.html (get the one with JVM)

Next you'll need to setup some stuff in Matlab. With Version 6 and later of Matlab, direct calls to Java are supported from the command line mode of Matlab. Thus, Matlab uses the native Java RBNB API directly.

1. Copy C:\Program Files\RBNB_V2.1\bin\rbnb.jar into C:\MATLAB6p5\java\jar\toolbox\

2. Add this line:
   $matlabroot/java/jar/toolbox/rbnb.jar

   into C:\MATLAB6p5\toolbox\local\classpath.txt

3. Copy the directory C:\Program Files\RBNB_V2.1\Matlab into C:\MATLAB6p5\toolbox\. DO NOT replace the directory already there but rename the copied directory RBNB.

4. Add C:\MATLAB6p5\toolbox\RBNB to the Matlab path ("File…", "Set Path…").

After that is all setup it's only a matter of downloading and extracting the client software.

Sapphire Support:

There are a couple of ways to tell if Sapphire is currently overhead. First you can download a demo version of Nova and run that to track

Sapphire. The other way is to use the satellite tracking built into SapphireTerm. This basically requires that you have the Predict, AtomTime, and http_get bundled zip file extracted to the C:\predict22 directory. This is nice because you can map out where Sapphire is right in a Matlab figure window. Below is an example of what you should see when you run simpleTermv3 in the extended mode of operation- meaning it will pop up a world map tracking Sapphire location, update keplerian elements on startup, and sync the time periodically. There is also a default mode of operation that doesn't have the frills just mentioned. To access this mode just type "b=simpleTermv3('129.210.19.118:3333')".

```
b=simpleTermv3('129.210.19.118:3333','e')
                       [simpleTerm v3.0]
----WELCOME TO THE GATEWAY OF REMOTE SATELLITE COMMUNICATIONS----
Connecting to 129.210.19.118:3333 ...
Connected.
Syncing time...
Updating Keps...
Done.
```

If all goes well when trying to connect to Sapphire (by typing "c ke6qmd") you should see the following:

```
c ke6qmd
c ke6qmd
cmd:*** CONNECTED to KE6QMD
Your passkey is 889916323.
:admin
144 165 227 059 054 084 112 205 066
Welcome to Sapphire.
? for help.
os time
os time
04:22:15 1/4/1995
…
```

After you get the passkey type ":admin" to automatically login with administrator privileges. After you are logged in start calling Sapphire commands.

Appendix J

Section J1: Use Case

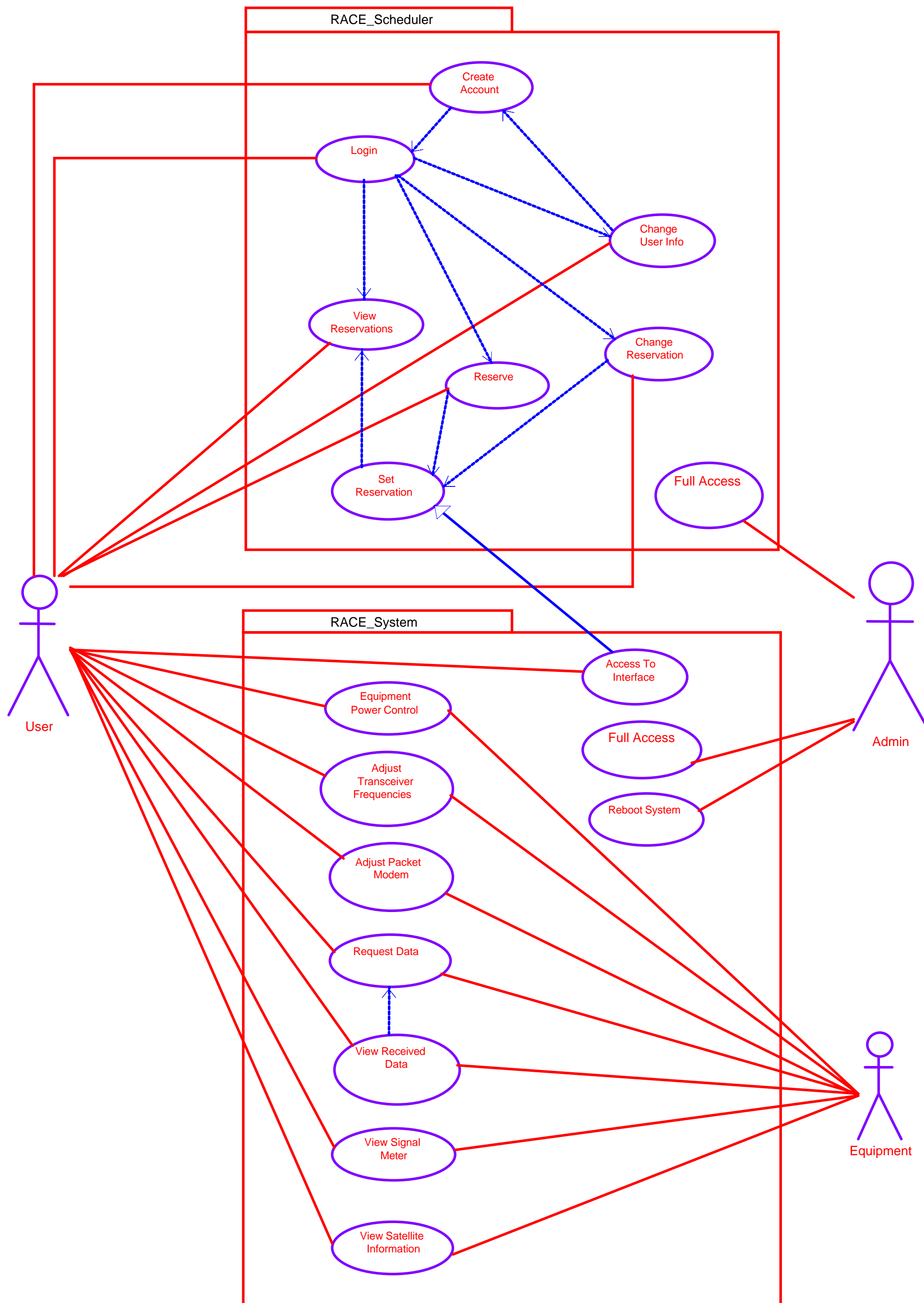

Figure J-1: Use Case Diagram

Section J1: Use Case Descriptions

The Use Case Description depicts how different users interact with the system.

It is described as follows:

Users: The users are those who will use the RACE System.

- ADMIN
  - The administrator has full access to the RACE system

- USER
  - The user is the person registered with the system. He or she can be located anywhere around the world.

Equipment: The equipment is the physical equipment used in the RACE system.

- EQUIPMENT**:**
  - The equipment handles user requests (such as adjusting equipment) as well as communication with the satellite (such as sending and receiving data).

  Equipment includes:
  - Server: *the server will provide the link between the user and equipment. It will also house the database.*

  - Packet Modem: *the packet modem will handle the data packets that are sent to and received by the satellite.*

  - CI-5 Level Converter: *the converter is used to make the transceiver controllable from a computer.*

  - Transceiver: *the transceiver receives data from the modem, converts it to radio waves, and sends through the antenna. It also receives radio waves from the antenna, which is decoded by the packet modem.*

  - Antenna: *the antenna is used to send signals to and receive signals from the satellite.*

  - Power strips: *the power strips are used to supply power to the equipment*

  - SASI Satellite Tracker: *the SASI satellite tracker enables the antenna to be controlled by calculations.*

  - Rotator Controller: *the rotator controller is used to control the rotation of the satellite.*

  - Rotator: *the rotator is the physical rotation equipment on the antenna.*

  - Remote Reboot: *the remote reboot allows the system to be rebooted from a remote location.*

Race System: The Race System describes how the user can use the system.

- **ACCESS TO SYSTEM**
  - User access to the RACE System is granted only by a valid and confirmed reservation time, which depends upon the scheduled time in the RACE Scheduler System.

- **ADJUST PACKET MODEM**
  - This allows settings on the packet modem to be changed. Through the RACE System, the user can adjust settings of the packet modem, including powering it on or off.

- **ADJUST TRANCEIVER FREQUENCIES**
  - This allows the transceiver controls to be changed. Through the RACE System, the user can adjust transceiver frequencies and settings.

- **EQUIPMENT POWER CONTROL**
  - This allows system components to be turned on or off. Through the RACE System, the user can power on or off equipment such as the packet modem and transceiver.

- **FULL ACCESS**
  - Full Access is granted to the Administrator.

- **REBOOT SYSTEM**
  - This reboots the system. Access to Rebooting the System is handled only by the Administrator.

- **REQUEST DATA**
  - Commands can be sent to the satellite to retrieve desired data. User commands and requested data are sent through the RACE system and are handled by the equipment, which are then to the satellite.

- **VIEW RECEIVED DATA**
  - As a result of a particular command sent to the satellite, the received data is displayed. After it has received responses from the satellite, the equipment displays the user-requested data.

- **VIEW SATELLITE INFORMATION**
  - This displays the current information about the satellite, such as azimuth and elevation. After gathering information (such as position in space) from the satellite, the equipment sends information about the satellite to the server-- which is displayed by the RACE system.

- **VIEW SIGNAL METER**
  - This displays the current connection status and signal strength to the satellite. The equipment sends information about the signal strength to the server-- which is displayed by the RACE system.

- **EQUIPMENT POWER CONTROL**
  - Through the RACE System, the user can power on or off equipment such as the packet modem and transceiver.

- **ADJUST TRANCEIVER FREQUENCIES**
  - Through the RACE System, the user can adjust transceiver frequencies and settings.

- **ADJUST PACKET MODEM**
  - Through the RACE System, the user adjusts settings of the packet modem, including powering it on or off.

- **REQUEST DATA**
  - User commands and requested data are sent through the RACE System are handled by the equipment, which sends them to the satellite.

- **VIEW RECEIVED DATA**
  - After it has received responses from the satellite, the equipment displays the user-requested data.

- **VIEW SIGNAL METER**
  - The equipment sends information about the signal strength to the server-- which is displayed by the RACE system.

- **VIEW SATELLITE INFORMATION**
  - After gathering information (such as position in space) from the satellite, the equipment sends information about the satellite to the server-- which is displayed by the RACE system.

RACE Scheduler[25]: The RACE scheduler handles reservations to the system. It also is used to determine access to the RACE System

[25] Implemented by the RACE 2002-2003 team. Please see RACE 2002-2003 Senior Thesis for further information and specifications on the RACE scheduler.
http://www.cse.scu.edu/send.cgi?srprojects/2003/COEN-2003-PROJECT-25.pdf