Final Report for
*The Boeing Corporation*

2005-2006 GPS IIF Control Segment Clinic
Final Report

May 7, 2006

**Team Members**
- Chris Dahlberg
- Andrew Kim
- Chris Erickson
- Marshall Pierce (Project Manager)

**Advisor**
- Professor Michael Erlinger

**Liaisons**
- Darryl Nakata ’86
- Ryan Ripken
Abstract

To maintain accuracy, GPS satellites require regular corrections to their broadcast orbital parameters. An extensive network of ground antennas and control stations throughout the world calculate the proper orbital parameters of the satellites. These corrected parameters are then transmitted to the satellites. Currently, this process occurs at least once a day. Raising the update frequency would increase the accuracy of GPS and allow increased functionality to be piggybacked on the GPS satellites. However, additional updates would increase network traffic between ground stations, with unknown consequences. To test the impact of such changes on network load, the clinic team is developing a simulation framework using inexpensive PCs to represent the various nodes in the ground network. Presently, link parameters such as latency or maximum bandwidth can be applied, traffic patterns can be generated, and the resulting network load monitored. There are also GUI tools to set up the network topology and configure the traffic flow that runs on the network.
# Contents

Abstract iii

1 Introduction 1
   1.1 Background ........................................... 1
   1.2 Task .................................................. 2
   1.3 Overview ............................................. 3

2 System Usage Guidelines 7
   2.1 Hardware ............................................. 7
   2.2 Preparation .......................................... 7
   2.3 Running a Simulation ................................. 8

3 Setup and Configuration 9
   3.1 Operating System .................................... 9
   3.2 Runtime Configuration ............................... 11

4 Traffic Generation 13
   4.1 Outgoing Traffic ................................... 13
   4.2 Responding to Incoming Traffic ..................... 15
   4.3 Controlling Traffic Events ......................... 16

5 Configuration Parsing 19
   5.1 REXML ................................................. 19
   5.2 Traffic Event XML Parser ............................ 19
   5.3 Link-Configuration XML Parser ...................... 22

6 Graphical User Interface 25
   6.1 GUI Toolkit .......................................... 25
   6.2 Current GUI Tools ................................. 26
## 7 Spectrum and Network Management

- **7.1 Introduction** ........................................... 31
- **7.2 System Requirements** ................................. 31
- **7.3 Installation Process** ................................. 32
- **7.4 SpectroGraph** ...................................... 33
- **7.5 AutoDiscovery** ................................... 33
- **7.6 Device Topology View** ............................ 34
- **7.7 MIB Tools** ......................................... 34
- **7.8 Spectrum vs. OneClick** ......................... 37
- **7.9 Review** ........................................... 40

## A Spectrum Quick Reference

- **A.1 Starting Spectrum** ............................... 43
- **A.2 SpectroGRAPH: Initial network setup** ........ 44
- **A.3 Useful Tools** .................................... 46
- **A.4 Alarms and Alarm Manager** .................. 50
- **A.5 MIB Tools** ....................................... 51

## B System Setup

- **B.1 Overview** ........................................ 55
- **B.2 Server Setup** .................................... 55
- **B.3 Client Setup** ..................................... 62
List of Figures

1.1 Topology of the simulation network .......................... 3
1.2 Example rack ............................................... 4

4.1 UML diagram of the Traffic classes .......................... 14
4.2 TRAFFICGENERATOR control flowchart. Rectangles are data structures, and circles are long-running loops. .............. 16

5.1 XML Configuration flow ...................................... 20

6.1 Link Configuration GUI ...................................... 27
6.2 Traffic Configuration GUI .................................... 28
6.3 Simulation Control GUI ....................................... 29

7.1 The device topology view for a node connected to a 3Com switch. .................................................. 35
7.2 MIB Tools view ................................................ 36
7.3 Graph of ifOutOctets identifier from a node ................. 38
7.4 SpectroGRAPH User Interface ................................. 39
7.5 OneClick User Interface ...................................... 40
7.6 View of Live Links between the nodes and the 3com switch 42

A.1 Spectrum Control Panel ...................................... 43
A.2 SpectroGRAPH Normal View ................................. 44
A.3 AutoDiscovery ................................................ 45
A.4 Model Information ........................................... 46
A.5 SpectroSERVER Performance ................................. 47
A.6 Device Performance .......................................... 48
A.7 SpectroSERVER Landscape ................................... 48
A.8 Device Topology .............................................. 49
A.9 Alarm Manager ................................................. 50
A.10 MIB Tools ........................................ 51
A.11 Bandwidth Graph ................................. 52
Chapter 1

Introduction

1.1 Background

The Global Positioning System, or GPS, has been widely adopted not only by the US military forces but also by the civilian population – hand-held receivers are inexpensive and plentiful, and real-time navigational systems for cars are now relatively common-place. This component of the GPS system is widely understood: place a receiver in view of a sufficiently large swath of sky and, after it does some trigonometry, its user can find their current position to within a few meters. What is not generally well-known is that there is a complex terrestrial network that monitors and updates the satellites. This network is necessitated by the fact that the satellites in the GPS constellation are, in fact, rather simplistic – they carry on-board an accurate timepiece and just enough sophistication to store their current orbital parameters and to broadcast that data to anyone who cares to listen. Additionally, they are able to monitor the health of various on-board systems. However, their orbital parameters are never known precisely and the satellites themselves have no way of detecting when their actual position and predicted position diverge. The network of monitor stations counteract this constant decay of the accuracy of the orbital parameter information. Without this network, GPS would soon be so inaccurate as to be effectively useless.

The ground network’s purpose is to alleviate the positioning problem. Its structure is divided into three kinds of installations: Monitoring Station (MS), Ground Antenna (GA), and Master Control Station (MCS). The MSs are at fixed positions and constantly receive GPS positioning data. The GA sites have an antenna capable of bidirectional communication with the
satellites for telemetry and command and control functions. The MCS is where the actual update computations are done: using the data from all the MS sites, the MCS generates correction information for each satellite, which is then sent out through the GAs to each satellite.

1.2 Task

This system works reliably, but in its current form, the satellites are only updated once every twenty-four hours, which consequently implies that the GPS signal is highly accurate only once a day. Having a more consistently accurate GPS system is clearly desirable. However, improving a system like GPS is more difficult than most other system upgrades: downtime is simply not tolerable on a system as critical as GPS. The current method for upgrading is to build an entirely separate network, test it thoroughly, and then move to the new network. Needless to say, this is expensive, especially if the changes do not end up being suitable for the communications load.

Understandably, Boeing would like to have a less expensive way to experiment with the GPS ground network capacity and network architecture, and has assigned this task to the clinic team. We are to provide a flexible, transparent simulation environment that allows users to specify the traffic that should flow across various sections of the network. In addition, we are to be able to monitor that traffic using the Spectrum network management package. There are several distinct problems that were addressed in order to provide such an environment:

- Setting up the physical hardware on which the simulations will run
- Maintaining a consistent operating system (OS) setup on each computer involved in the simulation
- Generating link configurations (such as a set of firewall rules) for each node
- Generating the desired traffic at the specified times
- Monitoring the generated traffic

The team has been working on this problem for two semesters. This document summarizes the completed project.
1.3 Overview

1.3.1 Physical Network

In the simulation framework, each site in the GPS ground system is represented by a single node (each of which is an inexpensive surplus PC provided by Boeing). This is an acceptable approximation because within a single site, communication is done with relatively high-speed Ethernet lines. The bottleneck that we are concerned with is the inter-site traffic, which passes over much slower lines. Co-located ground antennas (GAs)
and monitoring stations (MSs) have also been collapsed into single sites. This is reasonable as they are generally placed at the same physical site.

![Figure 1.2: Example rack](image)

In the actual GPS ground network there is an Alternate Master Control Station (AMCS), as well as external user sites. While the PCs are provided for simulating an AMCS and the user sites, we did not take these into consideration since including them would add significant complexity to our simulation without increasing its utility. Links in the diagram are physical 100Mbit Ethernet connections. These links are throttled in software to reflect the bandwidth limitations of the actual GPS links. Additionally, there is a physically separate administration network that connects every simulation node to two auxiliary machines (a general “admin” server used to run various utilities and a network monitoring system running Spectrum). The administrative network is used to distribute configuration information and to monitor traffic without interfering with the simulation.
1.3.2 Node Management

The OS distribution system investigated in the 2005 summer research project continues to be used to keep the nodes in a consistent, known OS configuration. This system makes it fast and easy to re-image all nodes from a common disk image using the University of Utah’s Frisbee tool. An additional distribution system is also in place that allows the nodes to rapidly acquire traffic and link configuration files from a server on the admin server without re-imaging the entire drive or manually copying the files.

1.3.3 Link Configuration

The Perl link configuration scripts used during the summer 2005 research project have been cleaned up and translated to Ruby, the implementation language chosen for this project. Superfluous configuration used for the inter-satellite communication in GPS3 has been removed.

1.3.4 Traffic Generator

The traffic generator has the required essential functions. It can dispatch traffic events at specified times and generate appropriate responses to incoming events.

1.3.5 Monitoring

SNMP information from all the nodes can be collected and viewed in Spectrum. Alarms can be set on various parameters and graphs of network utilization can be generated. Progress has also been made on exploring the capabilities of OneClick, an alternate interface to the Spectrum database.

1.3.6 Summary

The subsequent chapters of this document describe in detail each of the aspects that has been summarized above.
Chapter 2

System Usage Guidelines

This chapter gives a brief outline of how to use our tools from start to finish to run a simulation.

2.1 Hardware

First, some number of computers must be arranged and connected to follow the desired network topology. The simulation network should use the network cards in the PCI slots, and the admin network should use the network port on the motherboard. (Other configurations are possible but are likely to require additional preparation.) A system running Spectrum and an additional system running FreeBSD (the admin server) should be connected to the admin network.

2.2 Preparation

One of the computers for the simulation should be given a basic setup as described in Section B.3. It can then be imaged to the admin server by following the directions in Section B.2. Next, each of the simulation computers can be sent this image by running frisbeed as described in the above mentioned sections. Finally, messages should be appear on each of the machines describing their progress. When a machine is finished being imaged, it will reboot into the newly installed system and begin waiting to start a simulation.

While this is going on, the configuration files can be created. The first, the link configuration, is done with linkGUI.rb and the second is done
with trafficGUI.rb. Their use is detailed in Chapter 6.

The third piece of preparation is making sure the Spectrum server is watching the network. This requires SNMP, which is configured on the clients as described in Section B.3. How to configure Spectrum to poll the clients for information is described in Section A.5.

### 2.3 Running a Simulation

After the configuration files and the computers are ready, the simulation can be started from the admin server. Use the goGUI.rb tool to select the traffic and link configuration files, and to send out the proper configuration files to the nodes and start the simulation.

Alternately, to process the configuration files by hand, use the following procedure. After the configuration files and the computers are ready, the simulation can be started from the admin server. Go to the base code directory run the following commands to prepare and distribute the files to the simulation computers. (The words in all-caps are specific to your simulation. FILE.xml is a configuration file, DIR is the temporary directory used to store simulation files.)

```bash
# ruby parsers/linkToNative/LinkParserMain.rb 
  -c LINKS.xml -o DIR
# ruby parsers/allTrafficToPerIP/TrafficParserMain.rb 
  -c TRAFFIC.xml -o DIR
# ruby utility/distribution/signaller.rb 
  -c LINKS.xml -r DIR -h ADMINIP
```

And that’s it! The simulation should be starting at its scheduled time.
Chapter 3

Setup and Configuration

3.1 Operating System

3.1.1 Introduction

With several dozen node computers to maintain, some configuration automation is clearly necessary, since manual configuration is error-prone and slow. Frisbee\(^1\), a tool from the University of Utah’s Emulab project, lets the user create hard drive images and distribute them out to many nodes simultaneously. The basic principles of using Frisbee will be covered below, and details of using Frisbee may be found in the README\(^2\). That document should be consulted before using the system, as it can and will effectively erase the hard drive of any system it is misused on.

3.1.2 Creating an Image

The first step in creating an operating system (OS) image for the node computers is to set up one computer with the required OS and software packages. It is important to choose a hard drive partitioning scheme during OS installation that will fit onto the smallest hard drive that the image will be installed upon, or the image installation process will be unreliable at best (or simply fail outright).

Once the OS has been configured and the software has been installed, boot that computer from a burned copy of Frisbee CD ISO image\(^3\), which

---

\(^1\)http://www.cs.utah.edu/flux/papers/frisbee-usenix03-base.html
\(^2\)http://www.emulab.net/downloads/frisbee-README-20040312.txt
\(^3\)CD image available at http://www.emulab.net/software.php3
is a modified version of FreeBSD 4. The OS on the CD will ask for basic network setup and a root password to use. Once its network connection is set up, use another computer to log in remotely and run the Frisbee disk imager, which will leave a copy of the OS image on the computer that was used to log in. Make sure there is plenty of disk space available, as the OS images can be quite large (over 1 GB). Various levels of compression can be used, but we found level 4 to provide the best mix of speed and compression.

3.1.3 Distributing an Image

Image distribution from the server to the nodes can be done in two basic ways: broadcast/multicast and unicast. Unicast is more likely to work under unusual network circumstances (poor networking hardware or firewalls), but only is able to serve to one node at a time and should only be used as a debugging tool or if only one node needs imaging. Broadcast, or preferably multicast, can serve to multiple nodes simultaneously. Under broadcast or multicast, the nodes will listen to the requests that are being made by other nodes and avoid re-requesting blocks that other nodes have already requested in order to limit server load. This, in addition to other optimizations, helps Frisbee scale very well. Broadcast uses the IP broadcast address to send data, which causes a nontrivial amount of excess traffic on the network. Multicast is a more efficient distribution scheme, and is preferred over broadcast, but may not work well or at all with poor-quality network hardware.

Regardless of which distribution scheme is used, the OS image should be placed on a server computer that is accessible by the node computers. The server should then run the Frisbee daemon frisbeed. The node computers need to run the Frisbee client without touching the hard drive that the image will be written to. This can be done by booting from the Frisbee CD or other CD, by booting from another hard drive, or by netbooting the computers. The default method is to boot each node with the Frisbee CD and then set up a network connection and run the Frisbee client frisbee on each one. To avoid all this manual setup, we used a netboot system that boots node computers into a modified version of the OS on the Frisbee CD and uses DHCP to assign each node an IP address so that it can communicate with the server. The adjustments made to the Frisbee CD to make it netboot successfully are complicated, and are documented in Appendix B.
3.2 Runtime Configuration

Frisbee solves the OS maintenance issue, but there is still the issue of sending new configuration files to the nodes when a new simulation needs to be run. To automate this, we have a custom daemon that runs on each node and waits for commands from the administrative server. The daemon responds to a handful of commands that allow it to fetch, decompress, and run new files, as well as other miscellaneous commands. A simple interface has been written that takes the user-generated configuration files that define the simulation and transmits them to the nodes.
Chapter 4

Traffic Generation

4.1 Outgoing Traffic

4.1.1 Overview

There are two different types of outgoing traffic in the simulator. Sized traffic has a specified size and is sent as fast as is possible over the network link. This is intended to represent traffic generated when a node has completed a calculation and is sending a command or result to another node. There are also bandwidth-limited traffic events, which specify bandwidth and duration rather than a fixed size. They send out appropriately sized packets every fraction of a second to generate that bandwidth. These are intended to represent MSs and GAs forwarding information received from a satellite within the simulation. In our design, the information and functionality required to issue these traffic events have been encapsulated into a pair of traffic event classes which both descend from a common abstract base class. This design offers several advantages. It removes the duplication of common functions from both event types, bundles the required information so that it is conceptually easier to pass around internally, and allows the scheduling components to ignore the difference between a bandwidth event and a sized event by exploiting polymorphism.

4.1.2 Implementation

OUTEVENT is the common abstract base class for both of the specific output event types (see Figure 4.1 for a UML diagram). It stores data common to all outgoing events like destination IP, start time, etc. It also has the capability to connect to a remote socket over the appropriate interface and cre-
ate arbitrarily sized buffers containing the event’s key padded with zeros. Essentially, this class’s purpose is to avoid duplicating some common features and let other components avoid dealing with the difference between bandwidth-based traffic and sized traffic.

**SIZEDOUTEVENT** is the class representing a block of data that should be transferred from one node to another as fast as possible. When instructed to transmit, it creates an appropriately sized payload buffer and dispatches it. The OS’s TCP implementation takes care of fragmenting the message into packets and transmitting them, using the same process that occurs when actual GPS packets are transmitted.

**BANDWIDTHOUTEVENT** is the class representing a stream of traffic originating from a node at a constant rate for a period of time. When instructed to transmit, the node calculates the total amount of data that it should send by multiplying the bandwidth and duration. The node divides this traffic into equal size messages sent out every half second (this value is a constant set in the code and easily changed to match the dispatch rate in the actual GPS system). The TCP stack handles the rest of the transmission. Although **BANDWIDTHOUTEVENT** sends out the equivalent of a **SIZEDOUTEVENT** every interval, the implementation for **BANDWIDTHOUTEVENT** doesn’t use **SIZEDOUTEVENTS**. Most of the functionality for either class’ transmissions is placed in **OUTEVENT**’s payload creation function and dispatching these buffers is a simple library call. There would be essentially no code savings and a slight increase in complexity if **BANDWIDTHOUTEVENT** used **SIZEDOUTEVENT**.

---

**Figure 4.1: UML diagram of the Traffic classes**
4.2 Responding to Incoming Traffic

4.2.1 Overview

To allow greater flexibility and complexity in the simulation traffic patterns, we provide the ability for an event reaching a node to produce responses on that node. Each outgoing event includes a unique 2 byte key which the destination node uses to determine what responses, if any, should be generated. Each node has a list of INEVENTS associated with it. INEVENTS are pairs of keys and outgoing events. When a node receives a message it triggers any INEVENTS with matching keys. The response can be scheduled immediately when the outgoing event arrives, after the incoming message finishes, or with a configurable delay after either of these times.

4.2.2 Design

The response system has two classes (Figure 4.1). The LISTENER class is responsible for accepting incoming traffic and watching for keys. The INEVENT class is responsible for storing a key and any associated outgoing events, as well as scheduling the outgoing event on command.

The LISTENER class is the more complicated of the two. During initialization it is in an accepting state where new INEVENTS can be added to the list of responses. At this point, it is not accepting network connections. Once the start() function is called, INEVENTS can no longer be added and the LISTENER starts watching for TCP connections on the simulation port. When a connection is initialized, it spawns off a thread to handle it. That thread searches the incoming data for a key and tries to match it against its list of INEVENTS. If there is a match, it either immediately triggers the INEVENT or saves it to be triggered after the connection completes, depending on the type of INEVENT.

INEVENT is fairly simple. When told to dispatch its outgoing events, it loops through all of its stored responses, copying them and offsetting their start time with the current time. This is necessary because the traffic scheduling system works on absolute wallclock time and the outgoing events are stored with time offsets. The offset events are then handed over to the scheduling system. INEVENT also has the capability of writing itself and its associated outgoing events to XML. This is discussed further in the section on the traffic XML parser.
4.3 Controlling Traffic Events

Figure 4.2: TRAFFICGENERATOR control flowchart. Rectangles are data structures, and circles are long-running loops.

4.3.1 Overview

The traffic event classes encompass all the information and functionality needed for a specific stream of traffic, but do not provide any way to dispatch events at appropriate times. Consequently, there is a controller which encapsulates all current outgoing events and schedules them to start their traffic flows accordingly. The design of the controller was complicated by the fact that the data structure containing the outgoing events needed to be dynamic: since INEVENTS can trigger corresponding OUTEVENTS that can
be delayed until some future time, the controller needs to be able to handle events being inserted on the fly and adjust its scheduling appropriately.

4.3.2 Design

As can be seen in Figure 4.2, there are multiple threads accessing the event storage data structure. To avoid tricky thread-synchronization issues, thread-safety has been integrated into the data structure. (Leaving the thread synchronization up to the controller that accesses the data structure is error-prone and provides no performance gain.) The data structure needed to have fast insertion and removal, with the further condition that it should be easy to choose the proper event to handle next. The “priority queue” data structure was a good fit: it has $O(\log n)$ runtime for both insertion and removal, with the additional benefit that the top of the heap is always the item with the next activation time, so peeking at the next item has $O(1)$ runtime. While there were several implementations already available of priority queues for Ruby, none of them satisfied the three key conditions of being thread-safe, of using the proper heap algorithm, and of having a clean, usable interface. K. Kodama’s PQueue library was essentially undocumented, and is not threadsafe. Brian Schroeder’s PriorityQueue library is well documented, but uses an algorithm (namely, the Fibonacci heap) that is unsuited to our needs, and is not threadsafe. Consequently, a custom implementation (in the PRIORITY_QUEUE class) was written in Ruby that satisfied all three requirements. Threads can simply call methods on a PRIORITY_QUEUE object, and all locking is handled behind the scenes. Furthermore, the class can work on any data structure that supports the $<$ and $>$ operators. (In Ruby, this is easily achieved for any class by overriding the $<=$ method for the class and mixing in the “Comparable” module, which will then provide $<$, $>$, $<=$, etc.) The pure Ruby implementation was sufficiently fast to avoid needing to turn it into a C extension: heap operations take less than a millisecond, even on a slow machine. C extensions are harder to write and debug, and require more work to deploy.

While not having to worry about thread locking on the event queue makes the controller simpler and more reliable, there is still a fair amount of complexity involved in keeping track of all the various events. (Again, refer to Figure 4.2 for a visual depiction of the internal logic of the controller.) The traffic controller (implemented in the TRAFFIC_GENERATOR class) spends most of its time waiting for a mutex on the Notification Queue to be released. When that mutex is triggered, the controller pops a notification off of the queue, calls a helper method (runNextEvent), and resets
the mutex. The `runNextEvent` method checks the top item in the event queue. If the difference between that event’s start time and the current time is within a certain threshold (on the order of less than a second), the event is popped off of the queue and passed to the `runEvent` method, which spawns a thread that waits until precisely the right time, and then triggers that event. (Each outgoing event needs its own thread, since the controller needs to support multiple simultaneous traffic streams.) The `runNextEvent` method then checks the start time of the new top of the queue, which is guaranteed to be the event with the next start time by the properties of the priority queue data structure. If the new top event’s start time is within that same threshold of the current time, the process repeats. When `runNextEvent` eventually reaches an event whose start time is far enough away from the current time to not fall within the threshold, it starts a timer thread with an expiration time of the time remaining until that event needs to fire, less a certain offset (to allow for processing time). When that timer thread expires, it places a notification on the notification queue, which will unlock the main loop, which subsequently calls `runNextEvent`, etc.

Of course, all that intricacy would be unnecessary if all outgoing events could be specified in the configuration: simply create a sorted linked list of events, and pop events off as needed. However, that is not the case. As can be seen in Figure 4.2, the `LISTENER` can add outgoing events to the controller dynamically through another method (`handleNewEvent`). That method is quite similar to the `runNextEvent` method described above in how it examines the new event’s start time. If the event’s start time is within the threshold, the method starts a controller thread with `runEvent` to trigger that event at the appropriate time, just as `runNextEvent` did. If the event’s start time falls outside the threshold, a timer thread is started to eventually alert the main loop, just as in `runNextEvent`, and the event is inserted into the event queue.

### 4.3.3 Summary

The `TrafficGenerator`’s extensive locking and reordering functionality allows `LISTENERS` to add new `OUTEvents` to the existing queue on the fly, while maintaining the proper order of queued `OUTEvents`. This centralized scheduling allows the traffic classes to remain relatively small and simple.
Chapter 5

Configuration Parsing

5.1 REXML

Throughout the simulator, there are data files that need to be stored on disk to be passed to the nodes or saved between execution runs. We’ve chosen to store these files in an XML format. XML is a plain text, flexible data storage format that is straightforward, human readable, and human modifiable. Using XML allowed us to create files to test the bulk of the simulator while still evaluating various GUI options and allows bypassing GUI tools if necessary. Ultimately, the XML format is too verbose for an operator to conveniently use for a large simulation, so a graphical tool to generate it was needed, but raw XML is a good intermediate step. To parse and output XML, we’re using the REXML parser\(^1\). This parser is freely available and provides an easy to work with API. XML documents are represented as a series of nested nodes with properties at each level. They can be selected and accessed using syntax similar to Ruby’s native container accessing syntax, so the meaning of the parser code is apparent to programmers familiar with Ruby.

5.2 Traffic Event XML Parser

5.2.1 Overview

To initially load the traffic information and to store it when moving it from the admin server to the test nodes, we needed a way to translate traffic events from internal data structures to a format on disk. One complication

\(^1\)http://www.germane-software.com/software/XML/rexml/
is that to ease configuration, the timing of events is initially specified as an offset from the start of the simulation. This makes it difficult to start the simulation at the same time across all nodes, since it can take a variable amount of time for a signal to propagate from the admin server to each node. However, we have exploited an existing synchronization protocol called NTP\(^2\). NTP is intended to very accurately synchronize clocks across variable speed network links. We can use NTP to synchronize clocks on each node and then specify the timing of simulation events in terms of a wall clock time to make them occur at the correct relative time.

To do this, we process the original traffic specification file on the admin server. The parser selects a base time far enough in the future for all the configuration files to transfer and replaces the time offset originally specified with the base time plus the offset. This way, the relative timing of events on different nodes is preserved.

The parser also splits the original single traffic file into a traffic file for each node. The per-node traffic files only contain those events either outgoing from or incoming to that node. This simplifies on-node processing,

\(^2\)http://www.ntp.org/
since the node no longer needs to separate personally relevant events from
irrelevant ones.

This same parser is used again on the nodes to read in the per-node-files
after they have been transferred, since the format is identical.

5.2.2 Interface

The traffic configuration parser takes the single main traffic file produced
by the traffic configuration GUI as a command line option and outputs a
separate traffic file for each node.

5.2.3 Design

The parser consists of three main classes, a main parser class that ties ev-
erything together and two classes representing timed events and response
events. The event classes store parameters about their traffic event such
as the time, response key, size, duration, etc. They can also write an XML
representation of themselves to a REXML document object.

The OUTEVENT class is actually an abstract base class. The OUTEVENT
interface exposes two main functions to the outside world: writeXML and
an operator to compare two events based on their start times. writeXML
writes the XML description of the event to a REXML document and is used
by the parser to produce the traffic files for each node. OUTEVENT also
provides a function for descendant classes, writeCommonXML. This func-
tion writes common node properties (destination, source, etc.) to a REXML
node. It differs subtly from writeXML in that it appends to an already ex-
isting node, while writeXML creates an entirely new node.

There are two classes descending from this base class, SIZEDOUTEVENT
and BANDWIDTHOUTEVENT. These represent events that contain a fixed
amount of data being dispatched as soon as possible, or a constant rate
being sent for a fixed period of time, respectively. The writeXML functions
in each add a new XML node labeled with the event type to the document,
insert properties specific to that type (size for SIZEDOUTEVENT, bandwidth
for BANDWIDTHOUTEVENT, etc.). They then use writeCommonXML from
OUTEVENT to add properties common amongst all events, resulting in a
completed event node.

The main parser class itself is fairly straightforward. It has a pair of
functions, parseOutEvent and parseInEvent, that each take a REXML node
of the corresponding type (that is, OUTEVENT or INEVENT classes), extract
the parameters, and construct a Ruby object. The parser applies these functions to all the InEvent and OutEvent tags in the XML and assembles the constructed objects into an array to return. If the parser has been placed in “admin” mode, it also converts the event’s offset to an absolute time by adding the wallclock time the parser was initialized and a configurable constant to the event’s offset. This behavior is used during the parsing of the original single traffic file on the admin server.

If the parser is placed in “node” mode, it passes the start time through unchanged, which is proper behavior for the nodes: they assume that the wallclock times have already been calculated.

With these tools, the per-node splitting program is straightforward. It uses the parser to get all the events from the input traffic file. It then creates a new REXML document for each host and loops through all the events, signaling them to write their XML form to the document for their source host.

On the nodes, the parser is used to read in the traffic file and generate Ruby objects for the events. These events are then used to drive the simulation.

5.3 Link-Configuration XML Parser

5.3.1 Overview

In the GPS system, the physical links between various sites run over a wide variety of underlying technologies with very different performance characteristics. The bandwidth, latency, and packet loss characteristics can vary greatly between links. To simulate these parameters in our system, we use FreeBSD’s dummynet tools. Dummynet was originally developed for testing network protocols and allows adjusting the relevant parameters of the simulation nodes’ Ethernet links. Dummynet is controlled by a series of firewall rules set on the command line and has less-than-friendly syntax. Additionally, there are a number of other network configuration commands that need to be executed on each node to set up the network. To centralize the specification of all these parameters and allow integration with a GUI, we have developed a parser that converts a single XML file specifying the interface and node parameters for every node in a straightforward format into a set of per-node shell scripts that perform the actual network configuration commands for that node. This parser is based on the Perl parser from the summer 2005 project, but is implemented cleanly
in Ruby and doesn’t deal with the extra configuration needed to represent inter-satellite communications.

5.3.2 Interface

The link configuration parser takes a configuration file as a command line argument and then outputs a hosts file (to avoid the need to set up DNS), a DHCP configuration file, and a firewall configuration file for each node. The DHCP file and hosts file are put on the admin server. Each node gets a copy of the hosts file and its specific configuration script.

5.3.3 Design

The link configuration parser design is split into three components: a data structure holding information about an individual node, a data structure holding information about an interface, and a main parser component that does the actual work of reading in the XML and outputting the configuration rules.

The interface and node classes are very simple. The interface simply collects a set of interface parameters into a convenient bundle. The node is only marginally more complicated. It includes some per-node information like IP and MAC addresses, as well as facilities for associating a set of interfaces for that node. Beyond this, all the processing is carried out in the parser class.

The bulk of the parser’s intelligence lies in the parser class. This class is capable of reading in the link-description XML file and generating a set of Node and Interface objects corresponding to that file. Most of the work is done by the XML parsing library REXML. Once the internal state has been built, a set of other functions read the Node and Interface objects and generate the appropriate FreeBSD scripts. This separation makes the system more adaptable to change. Both the XML file and the FreeBSD scripts can be altered without affecting the other.
Chapter 6

Graphical User Interface

6.1 GUI Toolkit

In selecting a toolkit, several options were considered. A key goal was to use a native Ruby toolkit to allow the re-use of the parsers and support code already written. The available toolkits with Ruby bindings were Qt, wxWidgets, Fox, and Tk; web-based interfaces (based on Rails or similar toolkits) were briefly considered as well. Given the limitations of simple web-based dynamic graphics, the restricted portability of more complex web-based dynamic graphics such as Flash, the fundamentally single-user aspect of the system (which does not mesh well with the multi-user nature of web sites), and the desirability of a visually complex display of the network topology for the GUI essentially disqualified the web interface, so we limited our choices to the GUI toolkits.

We had several criteria that the GUI toolkit should satisfy:

- **Object Oriented**: The style of programming used in GUIs maps well to the OO paradigm; conversely, programming GUIs procedurally is slow and harder to debug.

- **Maturity**: The toolkit should have a wide selection of predefined widgets so we can spend more time on our custom code and avoid re-implementing basic functionality.

- **Documentation**: Clearly, the more documentation, the better. Ruby-specific documentation is preferable to generic documentation, and example code is a plus.
• Familiarity: Having a team member experienced with a toolkit makes development significantly smoother.

After reading the documentation for each toolkit and some third party experiences\footnote{1} with the available frameworks, the Qt framework appeared to be the best fit for our needs, but due to problems getting it to compile on FreeBSD, we moved on, since the advantages that Qt held over the other toolkits were not worth the extra time and effort getting it to build properly.

As for the rest of the toolkits, Tk was quickly eliminated as a choice: although mature, it is not object-oriented, and harder to work with than some of our other choices. WxWidgets produces pleasant interfaces and is cross-platform, but the Ruby interface to Wx is relatively recent, and Ruby-specific documentation is scarce. We settled on using FXRuby, due to its documentation, maturity, and good Ruby interface. In addition, one of the team members has experience using FXRuby.

6.2 Current GUI Tools

6.2.1 Link Configuration GUI

The current implementation uses two GUI tools, each producing an XML file describing an aspect of the simulation. The first tool lets the user configure the network topology. It allows the user to create and remove hosts, to configure host information (such as admin IP and MAC address), to specify the links that connect them, and to set the parameters of those links (bandwidth, latency, and so on). This is a simple GUI, created primarily to make it easier to work with the underlying format, and not necessarily to hide the underlying implementation details of how the network is actually laid out. It allows the user to understand the layout of the network at a glance, something that is difficult when given only a textual description of the topology.

To set up a network layout, the user double clicks on the blank panel at the top of the screen shown in Figure 6.1. A new node is created, using the parameters specified on the node tab of the palette found in the lower right. To modify the parameters of this or any other node, the user clicks on the

node. The detail panel in the lower left shifts to display information specific to this node. Editing this information changes the node’s properties, such as its administrative IP and MAC address, and whether it is network hardware or a node that will be active in the simulation. To create links between nodes, the user holds the shift key and drags between two nodes. A new link is created using the parameters on the link tab of the palette panel. Again, the link’s properties can be modified by clicking on the link and editing the fields in the detail panel. Once the topology has been established, it can be saved for future use using the save command in the file menu.

6.2.2 Traffic Configuration GUI

The second tool is for specifying the traffic events for a simulation (Figure 6.2). On the top of the screen is a timeline displaying the order of outgoing events from each node. Each node and its interfaces are listed vertically along the upper left. Below the timeline region, there is a panel to display information about existing events (lower left) and one to create new events (lower right). The user first selects what topology the traffic will run over by using the ‘open links’ command in the file menu. The GUI first reads the XML file generated by the previous tool and creates a row for each interface on the nodes. The user can then proceed to open a pre-saved traffic file or to begin creating new traffic events.
Once both a link configuration file and a traffic configuration file have been generated, they can be fed into the simulation control GUI (Figure 6.3). The user enters the path to each configuration file in the boxes at the top of the interface and enters the ip address of the admin server below. By clicking ‘go’, the tool will automatically invoke the appropriate tool sequence to separate the configuration into per-node files, transfer them to the node, and begin the simulation run.
Figure 6.3: Simulation Control GUI
Chapter 7

Spectrum and Network Management

7.1 Introduction

Spectrum is a network management tool owned by Aprisma and Computer Associates. Spectrum is used to monitor the health and performance of a network using SNMP (Simple Network Management Protocol). Boeing has Spectrum installed and running on their current network. However, Boeing does not utilize all the functionality available and would like to expand their knowledge of the software. The purpose of using Spectrum in this clinic project is both to use network management tools to monitor the generated traffic and to further explore the capabilities of the entire software package.

7.2 System Requirements

- Windows XP Pro
- 1 GiB RAM
- 40 GiB hard drive
- 2 NICs
7.3 Installation Process

The software package for Spectrum includes a CD for a Windows installation, a CD for a Solaris installation, and a CD for the OneClick software installation. OneClick is a web-based Java application with a modern user interface. The web-based Java interface allows OneClick to be used on any system with Java support. A User-Guide CD is also given to help with installation and operation.

It is recommended that the target machine be reformatted before installing Spectrum software. Starting with a clean slate prevents complications and allows for a clean installation. Also, the sole username for the Windows machine must be “spectrum” and must be an administrator account.

After reinstalling Windows XP, the machine needs the correct drivers for the hardware installed on the machine. If two network cards are present, Spectrum will bind to the IP address of the top network card. Windows places the network cards in an ordered list, which can be changed under Control Panel → Network Connections → Advanced Properties. Also under the advanced properties of TCP/IP of the chosen network card, under the DNS section, the “Register this connection’s addresses in DNS” and the “Use this connection’s DNS suffix in DNS registration” boxes need to be unchecked. Otherwise Spectrum will fail if a dynamic IP is being used for the network card.

To begin installation, insert the Spectrum Windows disk and enter the Spectrum extraction key. The installation process is fairly automated. Follow the instructions. The set up on the test network only requires a basic installation of Spectrum on one system. Spectrum can be set up with a distributed installation, but this is unnecessarily complicated. The username and password should be the same for the Spectrum login as the Windows login (i.e., “spectrum”). The landscape handle value entered was 4, though this number was chosen at random and has no bearing on the basic installation. Once the Spectrum installation is complete, run the Control Panel and boot the SpectroSERVER to check if everything starts up and shuts down properly.

After the SpectroSERVER is installed, install the Java Runtime Environment and Java SDK version 1.4.1. Before installing OneClick, the SpectroSERVER must be running. Also, in the SQL server, the admin database containing usernames and passwords must also be running. During the installation insert the proper keys, values and usernames. Once again, the username and password must be the same as the Windows login. After in-
installation, test to see if the OneClick application begins properly and press “Start Console” to enter the modeling console to ensure the program is operating.

Re-insert the Spectrum installation disk and install Crystal Reports while SpectroSERVER is still running. Insert the appropriate keys and the installation should go smoothly.

Once the entire software package is installed, there are two service packs which need to be added, which our liaisons provided for us. The Windows Service Pack 2 for Spectrum should be installed first. The installation for the service is very similar to the original installation. When that is done, start SpectroSERVER and install the OneClick Service Pack 2. This is also very similar to the original installation. After installing the service packs, ensure all software components are working properly.

### 7.4 SpectroGraph

SpectroGraph is the main tool for network management. In SpectroGraph, the network model is displayed with a visual interpretation of how the network is set up. Initially there are two views in SpectroGraph, the world view and the universe view. These represent containers of the network. With no network modeled, there should only be the SpectroSERVER node represented by an octagonal icon. From the SpectroGraph window many other utilities can be accessed.

The SpectroGraph view is also the view where the visual representation of the network can be edited. In edit mode, the icons can be arranged to visually represent the network. After modeling in AutoDiscovery, the icons for each node are usually without connections. In edit mode these connections can be manually added to reflect the links in the network by selecting two adds and adding a pipe from the toolbar. When a connection is manually added, the connection is unresolved and the connection is represented in a gray color. Resolving a connection is done in the device topology view.

### 7.5 AutoDiscovery

AutoDiscovery is a tool used to find the components of the network through various criteria. The utility has two stages. The first stage is the discovery of network components. There are 3 types of discovery configurations. The first is the IP list, where a list of IP addresses are given and Spectrum searches the network for nodes with those IP addresses. If the nodes are
configured with SNMP and the correct community string is given, then they will be discovered. If no SNMP is configured for the nodes, then the option to “discover pingables” will ping the IP addresses and wait for a response. The second discovery configuration is with an IP range. This discovery configuration is the same as the IP list but with a range of IP addresses. The third type of discovery configuration is by type. Our clinic project did not explore this discovery type because the IP addresses of our network are already known.

The second stage of the AutoDiscovery tool is modeling. The modeling configuration determines how the model is displayed on SpectroGraph. For our purposes, our modeling was very simple. The modeling for a node with SNMP is done by the default values.

AutoDiscovery never created the links for us between our 3com Switch and its attached nodes. The pipes between them were edited manually in SpectroGraph under the edit mode. This is done by creating a link between two nodes. The pipe created will be a gray color, meaning nothing but a link is possible between those nodes.

### 7.6 Device Topology View

In the Device Topology view of the interfaces (Figure 7.1), the connections between two nodes are established at the interface level. If two nodes have a pipe between them, then in the Device Topology while in edit mode, the node in the upper left corner can be dragged to the correct interface connection and established. This determines how the nodes communicate with each other. All of our nodes are connected to the 3com switch through their respective ports. This connection should be modeled under SpectroGraph as a gold-colored link.

Live links can be enabled where alarms and events about the connection are triggered in real time. This is done by right-clicking on a link and enabling live links. The link will turn green to represent a live link.

### 7.7 MIB Tools

MIB Tools is a SNMP management tool. SNMP is a network management protocol developed to access and modify data through a network with ASN-1 object identifiers. The object identifiers are organized in a hierarchy where each number represents a group and identifier (Figure 7.2). The SNMP protocol queries these identifiers and with certain access privileges
Figure 7.1: The device topology view for a node connected to a 3Com switch.
Figure 7.2: MIB Tools view
Spectrum vs. OneClick

is able to modify them, allowing remote management of a network from a central location.

Using SNMP, queries can be used to extract information regarding the node. For example, the sysDescr identifier under management → MIB-II → system returns the operating system running on the node. The number of packets or routing tables in the 3com switch can also be queried.

Some queries can be graphed by the MIB tools. For example, the ifOutOctets identifier can be graphed by using the button on the bottom right of the window. For graphing options, we have a pie chart, a bar graph or a line graph. Also, there is a checkbox to graph delta values, which means the first derivative of the value is graphed instead of the actual value. Figure 7.3 shows a delta value line graph for the ifOutOctets identifier. The graph represents the number of Octets leaving the node at a given time.

A very annoying “feature” in the graphing tool provided by Spectrum is that the realtime graphs are scaled in realtime. When data is being viewed in real time, the Y axis will change its range to accommodate spikes or dips in the data. This makes it extremely difficult to notice changes over time by just glancing at the graph, because in one span of time 10 KiBps could be at the top and later, 10 MiBps could be at the top and both would look the same unless the scale was carefully noted. Also there is no way to compare visually two different periods of data because as the graph is scrolled from left to right, the range scales along with it, making the average always in the middle of the graph. This feature renders the graphing tool in Spectrum almost completely useless, which is why OneClick was the focus of the second semester.

7.8 Spectrum vs. OneClick

OneClick differs from the Spectrum Package in a few ways. While OneClick uses the same information from the Spectrum database server, OneClick’s user interface (Figure 7.5) is entirely different. The user interface for Spectrum (Figure 7.4) is archaic and cumbersome, while OneClick’s web-based UI provides better usability and presentation.

A few key areas in which OneClick improves on Spectrum’s interface are:

- Different depths of the network and properties require a new window in Spectrum, which causes confusion and clutter. With OneClick, all levels of the network can be accessed through a tree hierarchy in the
Figure 7.3: Graph of ifOutOctets identifier from a node
subwindow on the left side. When navigating through the tree, the main display window changes accordingly.

- The properties window below the main display in OneClick is also designed for easy navigation and displaying important information about the network in a convenient location. The properties window is tabbed to display various useful properties of the network/node under observation. The OneClick GUI makes use of tabs, as well as allowing the user to place graphs or properties in separate windows so multiple windows of information can be viewed if desired.

- OneClick’s choice of icons, colors, and text provides a cleaner, more aesthetically pleasing appearance. The icons are more representative of nodes than the grey boxes used in Spectrum. The lighter background in OneClick’s windows provides more contrast for the colorful icons and text, which increases readability. Spectrum uses darker colors for the background, making text hard to read.
For these reasons, OneClick is our preferred interface. Spectrum may still be the choice for network management due to its extensive functionality. However, the interface in OneClick is more tailored to the style of current UI expectations and we believe it would be preferred by the users at Boeing.

7.9 Review

Spectrum is a powerful network management software. With so much functionality built into the software, it is difficult to figure out how to operate Spectrum through the current interface. Much of the information is hard to access and requires multiple window navigations.

The major fault of Spectrum is that the software is fragile. Small system changes to the system can cause Spectrum to mysteriously fail. We have re-installed the software numerous times due to simple but critical errors. Many of the problems we encountered are not repairable or have very poor
However, despite the problems, Spectrum is capable of managing a large scale network. The OneClick interface is more usable and is an aesthetically pleasing alternative, so we recommend its use.
Figure 7.6: View of Live Links between the nodes and the 3com switch
Appendix A

Spectrum Quick Reference

A.1 Starting Spectrum

To run the Spectrum software after proper installation, go to Start → Programs → Spectrum → Administration → Control Panel. This will open the panel shown in Figure A.1.

![Figure A.1: Spectrum Control Panel](image)

The Spectrum package is centered around a back-end database called
SpectroSERVER. SpectroSERVER keeps information concerning all the models, network status, and configuration. Without SpectroSERVER running, the other components in the Spectrum software will not run because communication with the SpectroSERVER is required. To start SpectroSERVER, simply click the Start SpectroSERVER button in the Control Panel (Figure A.1) and wait a few minutes. Also, it is recommended that you use the save option under the Database Administration panel when the SpectroSERVER is in a working state. To restore the administrative database to a previously saved state, press the restore button.

Once the SpectroSERVER is running, click the SpectroGRAPH button to launch the main GUI. The OneClick GUI can be reached by going Start → Programs → SPECTRUM OneClick console or by opening a web browser and opening the URL location http://localhost/spectrum/index.jsp and clicking the Start Console option.

A.2 SpectroGRAPH: Initial network setup

Initially the universe topology (Figure A.2) will just contain the SpectroSERVER. To start modeling the network, open AutoDiscover (Figure A.3) under Tools. To model your network, create a discovery and model configuration. You can use an IP list or range or model type, and Spectrum will scan the network to find those IP addresses or models. To have useful
results returned from the discovery it is helpful to have SNMP turned on for the nodes in your network. For a node with SNMP disabled, the discovery will return the node as **PINGABLE** which means the node is there, but no further communication besides pings is possible. With pings, up/down status can be obtained, which could be sufficient for some nodes.

![Figure A.3: AutoDiscovery](image)

Once the nodes in your network have been discovered, Spectrograph will include them in the topology. Most likely Spectrum will model the nodes independently. To model the connections between the appropriate nodes, the interfaces between two nodes have to be connected. First go into edit mode by clicking the edit button in the tool bar. To create a connection between two nodes, click one node and shift-click the second node. In the editing tool bar, there is a Connect Two Models button. This will connect two nodes together by creating a gray pipe. If you save the edit and go back to normal view and if the connection is valid, the pipe will appear a gold color. Right-click a node and go to DevTop → Interface to see the nodes’ interfaces. This Device Topology window will show the model type and which interfaces are turned on. Spectrum should automatically know which interfaces the nodes are linked by, but if not, the node will appear in the upper left box and can be dragged to the appropriate interface to link them together. To make the connection live, right click on the pipe and
enable live links.

In edit mode, the look of the network model can be manipulated. The colors can be changed and the location of the nodes can be dragged around so they more accurately represent the network. Also, in edit mode, node models can be erased or copied. Changes made in edit mode may not necessarily represent the actual network hardware. Now the network is modeled in Spectrum. If SNMP is running throughout you network, useful information can be gathered from the network through some provided tools.

A.3 Useful Tools

A.3.1 Model Information

The Model Information screen (Figure A.4) gives general information concerning the model, like system name, model type, model status, and SNMP polling information.
A.3.2 Performance View for the SpectroSERVER

This provides user server information such as memory and CPU utilization, and useful graphs pertaining to system and network performance variables. Navigating through the performance windows (Figure A.5), the health and status of the SpectroSERVER and network in general can be obtained. The CPU utilization shows how busy the SpectroSERVER is. The memory and disk utilization variables show how well the SpectroSERVER’s data is being handled and what kind of operations are using up the memory. Finally a Health Report can be logged with all the data stored in a file over a period of time.

A.3.3 Performance View

The Device Performance view (Figure A.6) gives packet information for a specific device such as percentage of frames received or percentage of frames lost, which can also be displayed in a graph. The data gathering starts when the view is opened. Once the view is closed, all data obtained in those graphs will be lost, as will any associated graphs. To keep track of data for long periods of time, the window needs to be kept open, or a report can be generated.
A.3.4 SpectroSERVER Topology View

The SpectroSERVER Topology View (Figure A.7) shows the user the structure of the SpectroSERVER. If the setup chosen for SpectroSERVER during installation was a distributed setup, then this might entail network management over the server itself. The setup used in the clinic project was based on one machine, resulting in a serene landscape.

A.3.5 Device Topology View

Device Topology view (Figure A.8) shows interfaces on a certain node and connections made to the node at the interface level. Right click on an interface to open a menu for interface options. More detailed information
on the interface is available, as well as a window with the threshold rates. In the threshold rate window, packet rate or utilization boundaries can be defined so that when they are breached an alarm event is thrown to the SpectroSERVER notifying the user of a problem. The off parameter for thresholds is the value below which the alarm will turn off.

**A.3.6 Device Interface View**

Device Interface view (Figure A.8) shows the status of the interfaces for that particular node. Right click on an interface to configure the interfaces thresholds. A threshold for an interface will set off an alarm if that threshold is exceeded. The menu for right-clicking an interface also lets you configure the interface and see other information.
A.3.7 Application View

Shows the various SNMP versions available and resources available to each type of SNMP query.

A.4 Alarms and Alarm Manager

Alarms are set off when certain events happen. In Alarm Manager (Figure A.9), the list of alarms that have not been cleared can be viewed. This list can be filtered for keywords or types of devices. The list can also be sorted by priority of alarms so the most urgent and critical alarms appear at the top.

Alarm Manager has a set of alarms built into the system. One of these alarms indicates when a link dies. This is categorized as a critical event. There are other built in alarms and events, though alarms can also be customized. Setting the threshold rate was previously discussed: when a threshold is breached, it will set off a minor alarm. To create custom alarms, a user can also use SpectroWatch. In SpectroWatch, custom alarms can be
made for almost any type of SNMP value with Boolean expressions. For more information on SpectroWatch, refer to the Spectrum documentation (file 0919).

When an alarm is activated, the event will appear in the list. Using the sub-window with the tabs, various information concerning each alarm can be viewed. Sounds can also be used to alert someone of an alarm. An alarm will give details concerning system information, date, and probable cause. Information for an alarm can be emailed to someone such as the system administrator or the owner of the node. Once an alarm has been acknowledged, it can then be cleared. Different policies in handling alarms can be configured with Spectrum. This is possible through SANM or Spectrum Alarm Notification Manager. SANM is a tool that allows automated handling of alarms in concordance with certain policies customized by the administrator. This feature was not used in the clinic project.

A.5 MIB Tools

Figure A.10: MIB Tools
MIB Tools (Figure A.10) is the feature that allows the user to make SNMP queries to a particular node. SNMP is a protocol which gathers information about a node and allows a user to access that information through a query. Some common information a user might want would be system name, uptime, or the amount of traffic that has flowed in and out or through the node. Routers can also be polled for router-specific information, like router tables, queue parameters, and network information. SNMP also allows editing of certain parameters. For more information on MIB-II, the data template for SNMP, refer to RFC1213 at http://www.ietf.org.

To use MIB Tools, we enter a node name or IP address of the machine. Once a connection is established we can use the MIB library to query the node for particular information. The tree of variables is a representation of the MIB library. Just choose a variable and click the Query button to query the node for that parameter. The queried information will appear in the sub-window if the query is supported. Graphs are also possible through MIB Tools. The little bar graph button in the lower right corner will allow graphs of certain parameters. The parameters must be integer values to be graphed.

To graph a parameter (Figure A.11), select the parameter in the information sub-window, then click the small graph button. In the pop-up window, choose what type of graph and how often Spectrum should poll the node.
for data. If the first derivative of the graph is desired then select the delta value checkbox. An example of a parameter where a real time delta graph would be useful is `ifOutOctets` under `Internet → mgmt → interfaces → ifTable → ifEntry`. This parameter is the number of octets which have left the interface. Both the total number of octets and the change in octets are useful information, but the convention for bandwidth is bytes transmitted per second, meaning the first derivative of the octets vs. time graph is desired. The graphs default to a 3D mode. To turn this off, right click and turn it off. All graphs are not labeled when opened. Graphs are customizable to the extent of labels and colors. The range is dependent on the window. The line graphs have the automatic rescaling feature so the range is always fit to the peak in the window at that time. Also, *a graph only polls for data when it is open, so once the window is closed, the graph is lost.* A log file can be stored, but the log file will only keep logging while the window is open.
Appendix B

System Setup

B.1 Overview

This is a detailed description of how we set up the server and created a client image. This is simply what worked for our needs and is in no way meant to be authoritative. Some details that are skipped in this guide are easily found in the FreeBSD Handbook (http://www.freebsd.org/handbook). Finally, some commands are too long to fit on one line on the page, but should be entered as one line on the computer. This has been explicitly noted on the entries that have been wrapped.

B.2 Server Setup

B.2.1 Introduction

This are the steps to create a server suitable for serving a netboot image and providing other administrative services. The following links were also helpful: http://www.freebsd.org/doc/en_US.ISO8859-1/articles/pxe/index.html, http://www.tnpi.biz/computing/freebsd/pxe-netboot.shtml, and http://daemonporn.com/wiki/index.php/PXE_install. Finally, Pawel Worach graciously spent several hours helping us get netbooting working properly.
B.2.2 Setup Procedure

1. Install FreeBSD, bring it up to the latest patch level\(^1\) (security is important on the admin server, since it will be serving as the firewall that acts as the external entrance into the private testbed network), install cvsup and portupgrade, run portupgrade -afr just to make sure that the buildworld was successful and didn't break any installed ports, and install isc-dhcp3-server, xorg, apache2, emacs, vim, svn, xml-simple (perl module), and ethereal. In the post-install configuration, we enabled linux binary compatibility, sshd, usbd and the blank screensaver, as well as enabling the 'fast' key repeat rate.

2. Download ROMP 0.2\(^2\) and run extconf.rb to create a makefile, then type ‘make install’. This will put romp_helper.so in /usr/local/lib/ruby/site_ruby/1.8/your_platform/. Unfortunately, romp.rb\(^3\) isn’t quite right – it uses an outdated method for getting object ids, and uses SOL_TCP as TCP’s protocol number for an IP header, instead of the correct IPPROTO_TCP. So, patch\(^4\) romp.rb by typing ‘patch romp.rb romp_patch’ once you’ve got the files in the same directory, then put your patched romp.rb in place.

3. Enable the internal admin IP address on whatever interface you’ve decided on (we’re using xl0) as the internal interface in /etc/rc.conf:

   
   ifconfig_xl0="inet 192.168.0.1 netmask 255.255.255.0"

4. Turn off sendmail and enable apache 2 in /etc/rc.conf:

   
   sendmail_enable="NONE"
   apache2_enable="YES"

5. Configure net/isc-dhcp3-server to start at boot by adding these lines to rc.conf:

   
   dhcpd_enable="YES"
   dhcpd_flags="-q"
   dhcpd_conf="/usr/local/etc/dhcpd.conf"

\(^1\)http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/makeworld.html
\(^2\)http://rubystuff.org/romp/
\(^3\)Found at /usr/local/lib/ruby/site_ruby/1.8/romp.rb
\(^4\)code/traffic_generator/romp-0.2/romp_patch in the repository
dhcpd_ifaces="xl0"

dhcpd_chuser_enable="YES"
dhcpd_withuser="dhcpd"
dhcpd_withgroup="dhcpd"

You’ll be adding in that dhcpd.conf file later, since it’s auto-generated by the scripts that read the XML link configuration file.

6. Adjust /etc/syslog.conf to write, for instance, local7.debug to /var/log/dhcpd.log. (Might as well make sure the ftp facility logs to somewhere you like – see below.)

    ftp.info /var/log/xferlog
    local7.debug /var/log/dhcpd.log

Adjust syslogd’s arguments so that it won’t open a network socket at all by adding this to /etc/rc.conf.

    syslogd_flags="-ss"

Restart syslogd to have it pick up any changes you made.

    # /etc/rc.d/syslogd restart

7. Enable inetd in /etc/rc.conf and make it only listen on the internal IP:

    inetd_enable="YES"
    inetd_flags="-wW -C 60 -a 192.168.0.1"

Enable tftp in /etc/inetd.conf, and make sure it’s chrooting to /tftp-boot. With the -l flag, it will log to the ftp facility, which by default goes to /var/log/xferlog. This should be entered on one line (it’s too long to fit on the page otherwise):

    tftp dgram udp wait root /usr/libexec/tftpd
    tftpd -l -s /tftpboot -u nobody

Add in a circular symlink to fix certain clients:
# mkdir /tftpboot
# cd /tftpboot
# ln -s . tftpboot

Note that our boxes end up requesting //pxeboot, so make sure that you can log in (with tftp <hostname>) and successfully get 'pxeboot', '/pxeboot', '//pxeboot', '/tftpboot/pxeboot', etc. Of course, it's fine if some of them fail if they are not requests that your specific PXE ROM uses when fetching pxeboot. (See below for where to get the pxeboot file.)

8. Add the following to /etc/rc.conf to enable NFS\(^5\):

```
rpcbind_enable="YES"
nfs_server_enable="YES"
mountd_flags="-r"
nfs_server_flags="-u -t -n 18 -h 192.168.0.1"
```

Edit /etc/exports (all on one line, as before):

```
/usr/local/netboot -ro -maproot=root -network 192.168.0.0
  -mask 255.255.255.0
```

Note that we need -maproot=root so that the client has permission to read spwd.db, because by default the nfs client is mapped to 'nobody' on the server, which can't read spwd.db. For security, make sure to use the -h flag to nfsd so that it binds to the private IP (192.168.0.1 for us) only. We're specifying a non-default number of daemons (-n 18) because we want to be able to serve a lot of concurrent netbooting clients.

And, of course, we need to make the directory where we'll be putting everything.

```
# mkdir /usr/local/netboot
```

9. Copy the pxeboot file from the frisbee ISO (assuming it’s mounted on /cdrom):

# cp /cdrom/boot/pxeboot /tftpboot/

10. Copy the filesystem from the cdrom to /usr/local/netboot using cpio to get all the special files like /dev/* correct:

    # cd /cdrom && find . | cpio -pdum /usr/local/netboot

11. Add a line to netboot/boot/loader.conf (there should only be user-config_script_load="YES" in there already)

    vfs.root.mountfrom="nfs;"

12. We need to set up the the conf/base and conf/default configuration that lets you define per-host changes for booting from the same base filesystem. We won’t be using the per-host configuration capability, but we will use base and default as a way to (fairly) safely make changes to configuration files.

    # cd /usr/local/netboot
    # mkdir -p conf/base/etc
    # cd conf
    # mkdir -p default/etc
    # cd /usr/local/netboot
    # find etc | cpio -pdum /usr/local/netboot/conf/base/

13. We need to have password databases created, or it complains it can’t find spwd.db on boot.

    # pwd_mkdb -d /usr/local/netboot/etc /usr/local/netboot/etc/master.passwd
    # pwd_mkdb -d /usr/local/netboot/conf/base/etc/ /usr/local/netboot/etc/master.passwd

    Note that right now, the root entry in master.passwd has a * in the password entry, which means you can’t log in as root. So:

    # cp conf/base/etc/master.passwd conf/default/etc/master.passwd

    Remove the * in root’s field in your new copy of master.passwd in default and recreate the password database.
# pwd_mkdb -d conf/default/etc conf/default/etc/master.passwd

14. We’re no longer booting / from a cd, so we need to change fstab.

    cp conf/base/etc/fstab conf/default/etc/fstab

Comment out the /dev/acd0 line, and insert one like this:

    192.168.0.1:/usr/local/netboot / nfs ro 0 0

15. Disable more cd-specific stuff:

    # cp conf/base/etc/rc.conf conf/default/etc/rc.conf

Disable the last two entries (root rw_mount and diskless mount) as well as sshd_enable. Edit /usr/local/netboot/etc/rc/rc and put a # in front of the line “EmulabCheckIPConfig”. We can’t do this one through the conf system because it’s executed before the conf/* is examined. (It should have been done in rc.local to begin with.)

16. Make /usr/local/netboot/usr/local/etc/rc.d/zzz.frisbee.sh (so it executes last), and make sure it’s chmod’d to 555. Put something like the following in it:

    #!/bin/sh
    case $1 in
        start)
            echo " ==> Frisbee client starting"
            /usr/local/bin/frisbee -m 239.0.0.147 -d -p 3564 /dev/ad0
            echo " ==> Frisbee client done, rebooting"
            shutdown -r now
        ;;
    esac

    The multicast group we’re using (239.0.0.147) is not magical – replace it at will if that group is already in use.

17. Adjust default/etc/dhclient.conf so that it only requests the host-name option:

    request host-name;
Turn on dhclient for the admin interface (xl0 for us) in default/etc/rc.conf:

```
ifconfig_xl0="DHCP"
```

Since we’re mounting all sorts of stuff over NFS, we can’t let dhclient bring xl0 down and up, but we do need the hostname information, or frisbee client complains about gethostbyname failing. So, use our custom dhclient script\(^6\) instead of the default one so that dhclient won’t reset the interface. The dhclient before FreeBSD 6.0 has various issues. (The one in 6.0 may have the same problems, but it has not been tested.) Put it in netboot/sbin/ and chmod it 555.

18. Download the frisbee tarball from emulab.net, and start frisbeed (use whatever path you need to get to frisbeed) once you have an image to serve (see the Frisbee README on getting an image). We have a small convenience wrapper (below). Put it in a file and chmod it 555.

```
#!/bin/sh

if test $1
then
  echo $1
else
  echo "supply a filename, fool!"
  exit 1
fi

./frisbee-snapshot-20040312/bin/frisbeed -d -W 80000000 \
-p 3564 -m 239.0.0.147 -i 192.168.0.1 $1
```

(enable debug output, port 3564, 80Mbit, use the interface that 192.168.0.1 is bound to, send to multicast address 239.0.0.147, serve the image passed as an argument to the script.) Note that frisbeed can (and should) be run as a non-privileged user. Also, frisbeed has a half-hour timeout, so if no requests arrive in 1800s, it will quit. Whether this is a feature, or just really annoying, we leave to the reader to decide. In any case, make sure the daemon is actually running if you’re troubleshooting.

---

\(^6\)frisbee/server_conf_files/netboot_stuff/dhclient-script in the repository
19. Setup NTP, since we want good clock-synchronization for the clients. Put this in /etc/ntp.conf (assuming you want to use 134.173.42.99 as your NTP server – which you probably don’t).

```
server 134.173.42.99
driftfile /var/db/ntp.drift

restrict default ignore
restrict 134.173.42.99
restrict 192.168.0.1 mask 255.255.255.0 nomodify notrap
```

Adjust rc.conf to start ntp:

```
ntpd_enable="YES"
ntpd_flags="-g -p /var/run/ntpd.pid"
```

20. Add the admin interfaces of the cluster to /etc/hosts and /usr/local/netboot/conf/default/etc/hosts. They can be copied over from the output of the perl script. For us, using output derived from a small test configuration, that looks like this:

```
192.168.0.11 S1.kepler3.com S1
```

Unless you have anything other than the standard localhost stuff (::1 and 127.0.0.1) in your /etc/hosts, you should be able to just copy over the generated hosts file.

### B.3 Client Setup

#### B.3.1 Introduction

This describes how we set up the client image for the nodes.

#### B.3.2 Setup Procedure

1. Boot from the FreeBSD 5.4 install CD (CD #1)

2. Select the Standard Installation option. It’s easiest if you use all of one hard drive for FreeBSD, so select the hard drive you want and press ‘A’ in the next screen to use ‘all’ of it. When setting up slices, we left / at 256 MB and gave 512 to swap, /var, and /tmp, leaving the rest for /usr.
3. Select the Developer distribution and agree to install ports. Before you leave that menu, scroll down to Custom and remove dict, doc, and GNU info, as we won’t be using those for client machines. (The man pages could be handy, and don’t take up too much space, so we left them in.)

After it’s done with copying files from the cd (or FTP, or whatever it was you used), it’ll ask you some post-install questions. We answered yes to sshd and configured the console to use a fast key repeat rate and blank the screen and set the time zone to PDT (since it’s almost summer in California). We also added linux binary compatibility, since it might come in handy.

4. Add a user, if you want. We added a non-privileged user in addition to root. (Make sure to add that user to ‘wheel’ if you want to be able to su to root from it.)

5. Bring up an interface so that we can access the net. Install cvsup (we used the port, though pkg_add -r cvsup-without-gui will also work) and use cvsup to grab the 5.4-release source (as well as the latest ports definitions). Do the standard buildworld, etc, procedure to get up to the latest patchlevel, then compile a custom kernel with IPFW and Dummynet (don’t forget IPFW logging and hz=1000). See the handbook and dummynet(4) for more information on the kernel options. See /usr/share/examples/cvsup/ for supfiles for src and ports. These are the options we added to the GENERIC kernel configuration (in addition to changing ident and commenting out unneeded cpu lines, of course):

```plaintext
options           IPFIREWALL
options           IPFIREWALL_VERBOSE
options           DUMMYNET
options           HZ=1000
```

6. Once we have freshly updated ports tree, install portupgrade, and run cd /usr/ports && make fetchindex && portversion -vL = to see if anything is out of date. If anything is out of date, run portupgrade -ar. Check /usr/ports/UPDATING to see if anything needs special treatment (like the perl 5.8.6 → 5.8.7 transition).

7. Install zsh, and give your users a good prompt in their respective .zshrc’s. This one isn’t bad for starters (the newline in PS1 is intentional):
fg_red="$(print '\%{\e[1;31m}')"
fg_green="$(print '\%{\e[1;32m}')"
fg_normal="$(print '\%{\e[0m}')"

export PS1="\%* [%(!.${fg_red}.${fg_green})%n${fg_normal}@%B%m%b:%6c] (%!) %# "

8. Install net-snmp, vim, emacs, ethereal, xorg (note that X complains loudly if a hostname is not set), ruby (if it wasn’t there before), gem (i.e. ruby gems), wget, and links from the ports. Use gem to install log4r. Follow the instructions in the server section to install ROMP.

9. Run portsclean and portsclean -DD to clean out working directories and unneeded distfiles.

10. Add to /etc/rc.conf:

   ifconfig_xl0="DHCP"
   firewall_enable="YES"
   firewall_logging="YES"
   firewall_script="/etc/ipfw.rules"
   sendmail_enable="NONE"
   snmpd_enable="YES"
   snmptrapd_enable="YES"
   ntpd_enable="YES"
   ntpd_flags="-g -p /var/run/ntpd.pid"
   gateway_enable="YES"

11. Put in /etc/ipfw.rules:

   ipfw add 65534 allow ip from any to any

12. Edit /etc/motd to be something descriptive like “GPS3 Client” and touch /etc/COPYRIGHT so that it won’t prepend the FreeBSD copyright message when the motd is displayed.

13. Make /usr/local/etc/snmp/ and edit snmpd.conf to be something like this:

   rocommunity <password> <Spectrum host>

14. Remove everything in /usr/src and /usr/obj (saves about 400MB)
15. Put `conf_client.rb` in `/root/gps3/conf/` and a simple script in `/usr/local/etc/rc.d` to load it at boot. This is the script we used in rc.d.

```bash
#!/bin/sh

case $1 in
  start)
    echo "Starting configuration client"
    cd /home/gps/conf
    su gps ./run_conf_client.sh
    ;;
  esac

We also used this convenience script in `/home/gps/conf/`.

```bash
#!/bin/sh

daemon -f ./conf_client.rb
```

16. Edit `/etc/ntp.conf`:

```
server 192.168.0.1
driftfile /var/db/ntp.drift

restrict default ignore
restrict 192.168.0.1
```

17. Configure sshd to allow login as root. We’re on a private network, so there’s hardly a security risk. `/etc/ssh/sshd_config`:

```
PermitRootLogin yes
AllowUsers root gps

Include any other users you want to be able to ssh in as in the AllowUsers line. (The non-privileged user we added was ‘gps’.)
```

18. You can now image the drive using the Frisbee CD (see Frisbee’s README for instructions). Note that you should make sure to stop ntpd and remove the driftfile (as well as making sure that `/etc/resolv.conf` is empty) before imaging.