PROPER HUMAN COMPUTER INTERFACING TO MULTICOMPUTER REMOTE MONITORING VIA ICMP AND SNMP

NAPS

Not Another Port Scanner

By:

N. R. Merizzi
P. Paszynski
M. V. Picheca
T. N. Tisdall

COMPUTER SCIENCE 4ZP6 PROJECT

Supervised by Dr. W. F. S. Poehlman
Associate Professor, Department of Computing and Software

McMaster University
Hamilton, Ontario L8S 4K1
2004

Abstract

According to SUN Computer Company, "the network *is* the computer"[1]. This belief is clearly supported by corporations that spend more resources on network security and infrastructure than on simply buying computers for associates. Not Another Port Scanner (NAPS) was developed to satisfy the growing demands of remote monitoring. Completed entirely in Java, the project provides an extensible, system independent and low maintenance package for any system administrator who needs assistance on monitoring their network.

The client and supervisor, Dr. Skip Poehlman, monitored and guided are efforts over the past ten months. Skip Poehlman, an associate professor at McMaster, completed his Masters in Science as well as his Doctorate in Computer Engineering and thus has a wide knowledge in computing. Ultimately, this product was geared to satisfy his needs as our primary client, but many in the future will be able to obtain and use this program for their own personal networks.

The group is made up of Nicholas Merizzi, Paul Paszynski, Matthew Picheca, and Tim Tisdall. All four of us are finishing our Computer Science degree at McMaster University. Throughout the year, many people helped us, especially by providing direction and testing our program. A special thanks to Mark Sibson, a graduate student at McMaster, for his effort and time this year.

Summary

The overall success of our work throughout the year surpassed what was expected of us. We managed to provide our results before our given due date with nearly 15,000 lines of code, organized into three major groupings, each grouping containing several modules. Therefore, by no means did we fall short of the course, our client, or even more importantly, our goals. We had set out initially to be able to monitor a collection of 50 to 75 computers located in our client's work lab. The result over the period of this year ended up being a full scaled monitoring tool that is capable of monitoring campus computers twenty-four hours a day. In this document we identify a number of areas where improvements can be done, which can be the basis for subsequent groups in the future.

[1] The slogan is purported to be coined by John Gage during a tour in China (http://www.wired.com/news/technology/0,1282,35539,00.html)

Table of Contents

List of Figures

List of Tables

Terms Used in This Document

Application Programming Interface (API) - An abstraction barrier between custom/extension code and a core, usually commercial, program. The goal of an API is to let you write programs that will not break when you upgrade the underlying system.

CHI (Computer Human Interface) – This field of computer science focuses on the methods of communicating an interface with human behaviour as best as possible. This deals with aspects such as color contrast, object positioning, and overall ease of usability for a user.

Client / Server - In the 1960s, computers were so expensive that each company could only have one. "The computer" ran one program at a time, typically reading instructions and data from punch cards. This was batch processing. In the 1970s, that computer was able to run several programs simultaneously, responding to users at interactive terminals. This was timesharing.. In the 1980s, companies could afford lots of computers. The big computers were designated servers and would wait for requests to come in from a network of client computers. The client computer might sit on a user's desktop and produce an informative graph of the information retrieved from the server. The overall architecture was referred to as client/server. Because of the high cost of designing, developing, and maintaining the programs that run on the client machines, US Corporate Enterprises is rapidly discarding this architecture in favour of the Intranet: Client machines run a simple Web browser and servers do more of the work required to extract the information.

Daemon - A program that runs in the background; that is, without user interaction, although it may listen for client requests.

Domain - A limited region or field marked by some specific property; for example, a field of knowledge, an industry, a specific job, an area of activity, a sphere of influence, or a range of interests. Generally, a system in which a particular set of rules, facts, or assumptions operates.

Domain Name Server (DNS) - A computer that translates hostnames to IP addresses on behalf of requesting clients.

Extensible Markup Language (XML) - is a simplified version of SGML with enhanced features for defining hyperlinks. SGML (Standard Generalized Markup Language) is a standard for how to specify a document markup language or tag set. SGML is not in itself a document language, but a description of how to specify one. As with SGML, it solves the trivial problem of defining syntax for exchanging structured information but doesn't do any of the hard work of getting users to agree on semantic structure.

File Transfer Protocol (FTP) - A protocol for transferring data files across a TCP/IP network.

Graphical User Interface (GUI) - is a program that lets the user interact with a computer system in a highly visual manner, with a minimum of typing. Graphical user interfaces usually require a high-resolution display and a pointing device, such as a computer mouse.

HCI (human-computer interaction) - is the study of how people interact with computers and to what extent computers are or are not developed for successful interaction with human beings.

Information technology (IT) - Includes all matters concerned with the furtherance of computer science and technology and with the design, development, installation, and implementation of information systems and applications [San Diego State University]. An information technology architecture is an integrated framework for acquiring and evolving IT to achieve strategic goals. It has both logical and technical components. Logical components include mission, functional and information requirements, system configurations, and information flows. Technical components include IT standards and rules that are used to implement the logical architecture.

Internet protocol (IP) - The standard that allows dissimilar hosts to connect to each other through the Internet. This protocol defines the IP datagram as the basic unit of information sent over the Internet. The IP datagram consists of an IP header followed by a message. [San Diego State University]

JUnit - JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.

MIB (Management Information Base) - All SNMP compliant devices contain a MIB which supplies the pertinent attributes of a device. Some attributes are fixed or "hard coded" in the MIB while others are dynamic values calculated by agent software running on the device.

Not Another Port Scanner (NAPS) – is the network-monitoring tool, has been built for the customer. This is the project goal of the group.

Object Oriented Programming (OOP) - A type of non-procedural programming where the emphasis is on data objects and their manipulation instead of processes.

OID (Object ID) – These are blueprints which provide one with the tools needed to load Management Information Base (MIB) files. These tools are allocated in a hierarchical manner, and are represented as strings and numbers.

Operating System (OS) - The most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

Process ID (PID) - Identifier of the process that instantiated it. This is the ID that uniquely identifies a particular process.

Serialization - The mapping of an N-dimensional data object into a 1-dimensional object so that, for example, the data object can be transmitted over the network as a 1-dimensional bitstream.

Simple Networking Management Protocol (SNMP) - is a standard TCP/IP protocol for network management. Network administrators use SNMP to monitor and map network availability, performance, and error rates.

Telnet - Internet standard protocol for remote login (terminal connection) service. Telnet allows a user at one site to interact with a remote timesharing system at another site as if the user's terminal were connected directly to the remote computer. [San Diego State University]

I. Requirements Analysis

The client's ultimate desire is to have an easy, fast, and efficient way of monitoring computers in his lab. His purpose for this is so that one day those networked computers can be used as a compiler farm either for his research or for future students. The client's most important need was the ability to always know the status of each node in the network. He was clear that he wanted a visual representation of the network in which he could then be able to proceed with gathering and setting information. Since the client is a professor, his need to manage these computers efficiently will be essential in the future.

The customer made clear a desire for many tools and functions, but some functions had higher priority then others. These requests were placed in order of priority and the lesser important properties were left as "add-ons" to be added by future developers. The obvious first need the client wanted was the ability to check the status of all the components in the lab, which included routers, servers, and networked workstations. If any problems occur in attempting to connect to the nodes, then proceed to writing the event to a log file, as well as providing both visual and audible alerts.

Another request which was left open ended, to be implemented if time permits, was for the customer to be able to modify certain aspects such as local time, hostname, and other modifiable OIDs. The obvious first level network check is using an ICMP ping. Due to network security issues in Java, our ping method was implemented by making an external call to the OS's ping command. SNMP was used to obtain and set the information remotely. This is what allowed us to provide the customer with higher levels of information and allow him to set, or get, variables. SNMP is also advantageous because of event traps. These traps, which monitor events such as system crashes, reboots, and incorrect logins, send visual alerts to the client program to notify the administrator.

An MIB browser was also added as an extra tool to provide complete control over monitoring these managed nodes. This feature provides a graphical tree-like interface for the user to search through and find the information desired.

Since the primary user is in a diverse environment where no one operating system is dominant, a flexible software package was needed. We chose Java for many reasons, but mainly so that the program would be OS independent. To keep the learning curve to a minimum, the location and placement of various functional objects was consistent, and followed a natural ordering. The main objective for the CHI was to provide quick access to many features, but not to overwhelm the user, therefore finding equilibrium between the two goals. The CHI represents clearly and rapidly the status of all the computers in a graphical way so the user can quickly see if the nodes are running or not.

Roles

Customer Roles

When deciding on the target audience for this program, it was quite clear that we had to design a product for a network administrator. The software itself needed to be able to handle small to midsize networks. This means, that any administrator that needs more control over their networks can use this tool. The program was designed not for novice users or users with little networking background, so although the client wanted the interface to remain simple, we did not allow this to come at the expense of functionality. The client's intentions are that one-day this project will be used as a compiler farm for either his research or future students. Our customer's most important need was the ability to always know the status of each node in the network. One of the biggest challenges to a network infrastructure manager is determining where the break is in the network, not actually fixing it. Thus, the customer using this monitoring tool will be able to do a better job by being able to identify these breaks much more quickly.

Designer Roles

The breakdown of the roles and responsibilities for each group member directly correlated to their specialization and expertise in the field of computer science. As a whole, the NAPS project can be broken down into three main modules. The first component is called the frontend of the application, which consisted of the interfacing with the user. The second main area is called the backend, which consisted of the interface with the SNMP library and the network probing. Of course, a bridge is required so the frontend and backend can communicate. This was accomplished by an intermediate data structure called the blackboard which makes up the third key component.

The backend and frontend are simplifications, which allowed for an easy division of work between the four group members. Although there were two members responsible for each major section, each individual was responsible for different aspects of the major parts.

For this application to have been developed correctly, each developer needed to have a good understanding of what the other team members were thinking. If one of the developers made a bad move in design, the other areas suffered greatly. Therefore, we held each other responsible for strong levels of communication in order to succeed. By achieving this, along with our other specified responsibilities, all the areas of this project were successfully covered. It came together as a complete solution rather than several disjointed parts.

Functionality

The main purpose of this software is to act as a monitoring tool for a set of computers located in the clients' research lab. The software is able to display the status of all computers that were inserted into it by the user. When the program starts for the first time, the user will have a blank window combined with a toolbar, status bar, and menu bar. The toolbar contains quick ways for the user to complete certain tasks like:

- Adding Node(s)
- Deleting Node (s)
- Help Menu
- MIB Browser
- Start/Stop Monitoring
- Saving A Mapping

Where as the menu bar provides everything the status bar provides and more such as functions for clearing a mapping, saving a mapping, and changing the preferences. Once the user has added nodes to be monitored they are visually represented by different types of icons depending on their status. Simple Green means the computer is up and running, but without SNMP. Green with the screen displaying "SNMP" will let the user know the node is alive and has the SNMP daemon running. The red screen shows that the remote node is down or cannot be reached.

The window will constantly be auto-refreshing itself when changes occur after the backend updates the blackboard. So if a certain node 'X' goes down, 'X' is automatically repainted to visually show this. This also causes an event to be written to the log file.

The user will have the ability to double click on any node to receive more information regarding its status. This node information window provides the following:

- Hostname
- Host uptime
- System description
- Total packets received on the interface
- PC Location
- Host IP Address

Now if SNMP is not available for that particular node then some of these values will obviously not be available. From here the user will have the option to open the MIB Browser on this node, or to update the information right away. The MIB browser, which can be reached in several ways, provides the user with the power to get and set specific information on remote nodes. The browser conforms to the RFC1213 standard, and once again presents the information in a user-friendly fashion.

The user, if they choose, will also be alerted to possible problems on the computers by liaising alerts when certain computers shutdown or have other errors that are reported by SNMP traps. These traps will report everything from computers shutting down to login failures.

The program is designed with extra information stored in tool tips. Even though the program is relatively easy to use, full explanations and examples are available through a well documented help menu. The color scheme of the program is easy on the eyes and has a modern look and feel to it which can be modified by the user.

Scope & Issue Tracking

After initially discussing desired features of the system with our client, we had to identify the significance of each requirement. In writing our initial extended project proposal at the beginning of the year, we classified them as follows: essentials, feasible but not essential, and areas of further development. The obvious three basic essential modules to satisfy his requirements were a blackboard, frontend, and backend. The frontend would provide a user friendly environment, while the backend would supply an efficient network probing system. Our level of responsibility was initially to provide the above three modules and then guarantee their functionality through thorough testing. From that point on, the other components would not be considered essential or areas of further development.

A few issues were raised that required us to meet and discuss certain aspects of the project. We logged these issues online and can all be reviewed. Table 1 displays a few of the issues that we needed to address along with their result.

Table 1 Major Issue Tracking Topics

Issue Topic	Resolution
Displaying Route Info	Our SNMP calls do not work for receiving tabled values correctly. Therefore we will not include this feature.
Ability to do Time Synchronization	Due to a lack of time we were unable to complete this task.
Dynamically changing detailed viewing	Due to a lack of time we were unable to complete this task.

Issue Topic	Resolution
Deleting ranges of Nodes	Because creating the maps became tedious we needed this function.
Should we make our program Skin able	This question was brought up in order to allow the user to enhance the frontend feel to his desire. We found that to be a key CHI issue we needed to resolve.
Cross-Platform Issue with Mac's	We wanted our software to be a true cross-platform program. Unfortunately we do not have access to a Mac PC enabling us to run and test our code on this platform.
Features to locate nodes and display node count Do we have the time?	This was brought up by our frontend designer. He had the ability to add these features and wanted to discuss with everyone to make sure it wouldn't set our other plans behind.
Instead of using embedded HTML in a custom window for help… Do we have the time/ability to interface with JavaHelp 2.0?	Towards the end of the project we discovered that Java had a package ready for us to interface with to implement our help menu for the user. The only unfortunate downfall was that the package is another overhead that we must learn. The group decided to use it and not cut the user-guide short. We wanted to provide a complete solution to our client.

The entries in Table 1 are considered the major issues that were brought up and needed to be discussed as a group. A complete issue tracking report including all bugs discovered, and who they were assigned to can be tracked down in our forum or through our email list archive, both available only online[2].

To clearly compare and contrast all aspects of the project that were initially talked about and what we completed, please refer to Table 2. This illustrates our status on all components and if we did not complete a certain task, it provides a brief reason why. All tasks were feasible in the sense that we were not limited in our design or choice of language, but the problem was time constraints that we had to follow.

[2] The forum is located at http://www.creativestudent.com/naps/ikonboard/ and the mailing list archive is located at http://server791.dnslive.net/pipermail/4zp6-naps_creativestudent.com/

Table 2 Categorization of the project components

Task	**Category**	**Status**	**Reason** **(if Not Complete)**
Frontend	Essential	Completed	----------------------
Backend	Essential	Completed	----------------------
blackboard	Essential	Completed	----------------------
SNMP Calls	Essential	Completed	----------------------
Ping Module	Essential	Completed	----------------------
Saving/Loading Mappings	Feasible	Completed	----------------------
MIB Browser	Feasible	Completed	----------------------
FTP	Further development	Not Completed	Time Constraint
Telnet	Further development	Not Completed	Time Constraint
E-mail (to inform the administrator with alerts)	Further development	Completed	----------------------
Route table of nodes	Further development	Not Completed	Time Constraint
Automatically discovering network topology	Further development	Not Completed	Time Constraint
Providing a PID table of current processes running on a node	Further development	Not Completed	Time Constraint
Installation package for Linux, UNIX, and MS Windows OS	Feasible	Partial Completion	Time Constraint
User Choice for SNMP values being kept in node info window	Feasible	Not Completed	Time Constraint
Skin-able Application	Feasible	Completed	----------------------
Synchronize Time	Feasible	Not Completed	Time Constraint
Printing Capabilities	Feasible	Partial Completion	Can print Log files, but not actual Mapping

As one can clearly see there were many areas, which were feasible, but were not deemed essential to this project. We had to prioritize and set a “Code-Stop” date which forced us to leave out certain features we knew we could do but were unable to complete within the remaining time. Components such as FTP, telnet, and routing table info were all aspects that would enhance the power of the software but unfortunately in order to have the proper time to test, and document we decided to avoid these features. We did however manage to complete several extra functions that were not initially expected of us. For example the ability to save and load network maps, add and delete ranges, and have a single executable file for easy distribution across any platform are some of the extra features we manage to complete. All these features provided functionality that will save much aggravation for the user, thus contributing to a user-friendly CHI.

Where we fell short was when we attempted to add the “extra” features, often referred to as “add-ons”, to the project. Features such as providing FTP and Telnet access would have been great to allow the user to connect to the machine. Another nice aspect would have been for the program to be able to scan for a certain valid subnet on its own and give the user the network mapping. From a frontend point of view we had wanted to provide more visual readings for the user. For example, instead of simply showing that a computer was alive we wanted to show visually that the node was alive and below it’s CPU’s current utilization by implementing a progress bar. To satisfy human behaviour at a higher level then what we had achieved we needed more signs, signals, and symbols to trigger certain behavioural responses from the user.

II. High Level Design

Overview of Possible Components for Monitoring

Research was done to determine what methods have been used by others in the past to monitor a network. The two major methods were using a client/daemon relationship, or using a client, daemon, and an intermediate server. The daemons could also be categorized into proprietary designs, (implementing a communication method only used by a particular client and only gathering specific information), or protocol compliant designs, (which implements a standardized protocol allowing several clients to utilize the information transmitted).

Client and Proprietary Daemons

Researching similar applications on the Internet, the majority of the programs work using a client/daemon relationship. The difference between this client/daemon relationship and the typical client/server relationship used in most network applications is that there is one client and many small “servers” or daemons; usually one on each machine in the network. Each daemon in the network sends information to the client and then the client displays the information in an easily readable format for the end-user. This information is either requested by the client or is broadcasted by the daemon, and the client passively captures that information. (Some systems use a combination of both depending on the content of the information. It’s useful to have the daemons broadcast important events and less important information is left to be sent only by request. This reduces the network traffic, but keeps the client up to date in an almost real-time fashion.)

The problems with this type of structure arise when there are several different platforms and OS’s attached to the network. Since each machine requires a running daemon to be monitored, a daemon must be created for each different platform.

The benefit of this architecture is that the daemons are designed to specifically obtain particular pieces of information, so the daemons can be optimized to provide just that information. This allows the daemons to typically be rather small and use low resources.

Client and SNMP agents (standardized protocol daemons)

Since monitoring nodes on a network is a common problem, standards have been created for communication between daemons and clients. This allows a person to use multiple clients which leads to different machines that can have different daemons. But they will all communicate in the same fashion due to the standard protocol. One such protocol is the Simple Network Management Protocol.

This is essentially the same as the previous design, except that there is a standard protocol to which the daemons have to adhere. The advantage of this system is that SNMP has been around for quite a while and there are many SNMP daemons already created for different OS's. The disadvantage is that a fully adherent SNMP agent tends to be rather large compared to simple daemons. This is since they are able to monitor a very large number of aspects on their network, and on the machine to which they are installed.

Client, Server, and Daemons

As networks get larger, a simple client/daemon relationship can create a large amount of network traffic (regardless of the daemon being proprietary or utilizing a standard protocol). This is especially true when the client is first run and it needs to gather all the information from the daemons to get a picture of the network. The initial time taken to collect all the data from all of the daemons can be quite lengthy and is usually the point at which the client/daemon communications are at their highest. One solution to this is to have intermediate servers that collect the information from the daemons on an ongoing basis. Then when a client is run, it simply retrieves all the information from one source only. This can be extremely beneficial in a network where there is more than one client being run at a time.

A decision was made to implement the "Client and SNMP agent" architecture described above to monitor the computers in the lab. The user's lab contains several different types of machines running a variety of operating systems, and as there is already many open source versions of the SNMP agent, as well as some of the operating systems are packaged with an SNMP daemons (ex. Microsoft Windows) we can concentrate on designing the client.

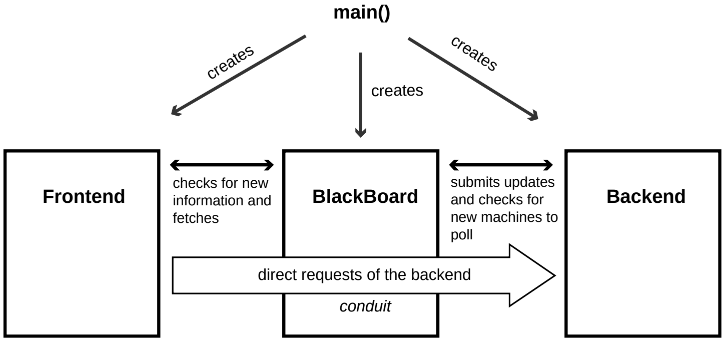

main()
creates
creates
creates
Frontend
checks for new information and fetches
BlackBoard
submits updates and checks for new machines to poll
Backend
direct requests of the backend
conduit

Figure 1 The three main parts of NAPS. The BlackBoard serves as an intermediate class to facilitate communication between the front end and the backend. The backend and front end implement threads so concurrent communication between the ends and the blackboard will occur.

Also, having a perpetually running machine acting as an intermediate server adds a level of complexity not required as the lab is relatively small. Such a server would also create a constant level of network traffic that may be undesirable when monitoring isn't necessary. It is also unlikely that more than one client will be running at any given time (although, the client will be able to handle such a situation if required), and the major benefits of an intermediate server is realized when there are many clients running.

Another benefit of the SNMP architecture is that there are many open source APIs available for various programming languages.

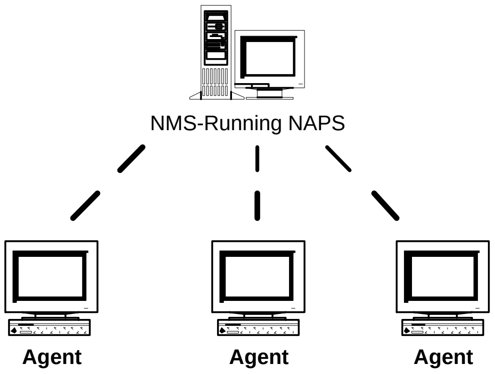
Module Interfaces

Simple Network Management Protocol

SNMP is defined in RFC 1157.[3] The protocol is used to manage a series of information stores called Management Information Bases or MIB's. The structure of these MIB's is described in RFC 1156.[4]

SNMP agent / SNMP API Interface

In an SNMP Network all monitoring nodes are referred to as managed devices, these network elements will be running SNMP agent software. This SNMP agent software will provide the network-management system (NMS) with the appropriate information. Since this is a standardized protocol, the SNMP API has to adhere to it when communicating with the SNMP agents. Since the method of communication, SNMP, is the interface between the SNMP API and the SNMP agents, and that has been standardized, there are no design decisions for this interface.



SNMP API / Backend Interface

The SNMP API consists of many classes that will cover everything from creating an SNMP object to SNMP vectors when needed. The core object that will be needed is the SNMPv1CommunciationInterface class. Once we have instantiated an instance of this class several methods are made available to us, for example, setMIBEntry, and getMIBEntry. The only class that will be calling this external library will be our

[3] ftp://ftp.rfc-editor.org/in-notes/rfc1157.txt
[4] ftp://ftp.rfc-editor.org/in-notes/rfc1156.txt

backend. Error handling during communication over the network will also be taken care of through this external interface.

Backend / Blackboard Interface

There are two major parts of the blackboard with respect to the backend. One part is a passive database of information that the backend is responsible for updating. The other part is an interface for the frontend to make direct requests (requests that are immediately fulfilled by the backend, although transparently) of the backend.

Having the frontend make requests to the backend through an interface, allows for any changes in the backend to be encapsulated. In other words, any change to the backend only requires a change to the blackboard class. The purpose of this direct request is to allow the MIB browser to make real-time requests for specific information and not require it to be stored in a database.

The majority of the information is obtained by the backend requesting LanNodes (for details see “Details of Blackboard Structure” below) from the blackboard, updating them via SNMP, and then returning them to the blackboard. This is accomplished with a simple “get next” type of method that allows the backend to iterate through all the nodes in the blackboard. (This may also be accomplished by implementing an *iterator* for the blackboard class.) The backend is responsible for keeping track of the time between polling, but the blackboard will be responsible for ensuring that all nodes are polled during one sequence of polling.

Blackboard / Frontend Interface

As mentioned in the description of the interface between the blackboard and the backend, the blackboard has a method for the frontend to make a direct request of the backend via the blackboard. This is a single method to request a particular piece of information (an OID) from a specific machine.

When a node is updated in the blackboard, an event is signalled to tell the frontend that there are nodes to be displayed. Like the backend, the frontend has a method of getting the “next” node to iterate through the nodes that require displaying (or require that the information in the GUI be updated). The frontend is also able to request specific nodes so if there is extended information to be gathered on a node (for instance, if a person were to click on a node and want to get further information) the blackboard can return that entire LanNode.

When a new node is to be added, the frontend makes a new instance of the LanNode class and then submits it to the blackboard. This is done in the same fashion that it returns already existing nodes that have been changed. The blackboard is responsible for distinguishing the difference between new nodes and already existing ones. When a new node is added then the blackboard will let the backend know there's a node that requires updating.

Details of Blackboard Structure

As mentioned before, the backend is responsible for updating the LanNode's in the blackboard. The blackboard provides some method for the backend to iterate through the elements ensuring that every node is gathered by the backend for updating during a particular polling event. There is a single method for submitting updated nodes to the blackboard as the blackboard is not concerned with who has updated a node, only where to put the submitted node (i.e. should it replace a node already in the structure, or is it a new one).

It has already been hinted at, but specifically, the method of adding a new node will simply be for a new instance of LanNode to be created and then submitted to the blackboard. The blackboard is then be able to distinguish a new node from an already existing node because every LanNode has a number tag that can only be changed by the blackboard. A LanNode submitted without a number is assumed to be a new node and assigned a number and a spot in the blackboard.

In order to avoid race conditions and collisions in the data structure, a "library" method will be used.[5] When either the backend or frontend has "checked-out" a node, no other part of the program will be able to take out the same node. The exception, though, is any part of the program can check out any node as long as it's in a "read-only" respect. This is done through a method that works the same as the regular get method, but does not check to see if the node has been checked out. It is the responsibility of the caller that that node is not then resubmitted to the blackboard for over-writing.

Since the blackboard is a stand-alone class, changes can be made to the data structure without affecting the overall function of the program. One such change that could be made at a later date is a priority queue to ensure that the most important nodes are queried first during a poll. Another change is the method of representing the data in the blackboard's data structure. Yet another change is making the blackboard more robust and not leaving any responsibility to calling objects to ensure that certain invariants are maintained (such as adding *assert* statements).

Event handling is used to notify the backend and frontend of work that needs to be done. That way neither the frontend nor the backend has to use a busy-wait method of polling the blackboard for new work to be done.

[5] This is also commonly known as a "write lock" in OS circles

Details of LanNode

Each node of the network has a record in the blackboard containing its IP address and a series of flags telling the blackboard whether the frontend or backend have checked-out that node and also whether or not the information contained in the node has been displayed by the frontend yet. (This record class is called a "LanNode".) Every record also contains a mapping of OID's to be requested and their corresponding information values. That mapping is stored in a Java HashMap which automatically expands if necessary and also allows every key-value to be returned in relatively the same length of time. Lastly, the LanNode contains a number (a unique id) to let the blackboard know where that record should be stored in the ArrayList.

When someone wishes to add a new node to be monitored on the frontend, the frontend creates a new instance of the LanNode class. The frontend then records the IP address of the node as well as a list of OID's (stored as keys in the HashMap) to monitor. Next, the frontend submits the record into the blackboard for the backend to pick up and update. The blackboard sets the flags in the node appropriately because the new LanNode does not yet contain an index into the ArrayList. The blackboard also assigns an index number and stores it in the ArrayList for the backend to find. When the backend does retrieve this particular LanNode, it fills in the values corresponding to the OID's in the HashMap in the same way as if the LanNode was an old record.

Currently the design specifications do not contain this feature, but if at a later date is becomes desirable to allow the backend to add nodes to the blackboard (perhaps as some sort of network discovery algorithm), then it could add them in the same fashion as the frontend. In order to facilitate this, and also to make loading and saving the network easier, the frontend is be able to dynamically handle the situation where it retrieves a node from the blackboard that it has not yet put in the display.

XML File Containing MIB Info

For the user to interact with nodes in question, they must be able to obtain OID's from a SNMP daemon running remotely. Since the list of possible MIB's available to a user is overwhelming, an XML file has been created with information on each OID in a tree organization that will be supported by this software. This allows the user to navigate through a listing of possible OID's with user friendly names and descriptions associated with each.

Once selected, the XML file containing the MIB's is opened, parsed, and then used to obtain the corresponding numeric OID value. To accomplish this, a tool called DOM (Document Object Model) is used to break the XML file into a programmable tree format. After this is done, the module then navigates through the trees looking for the

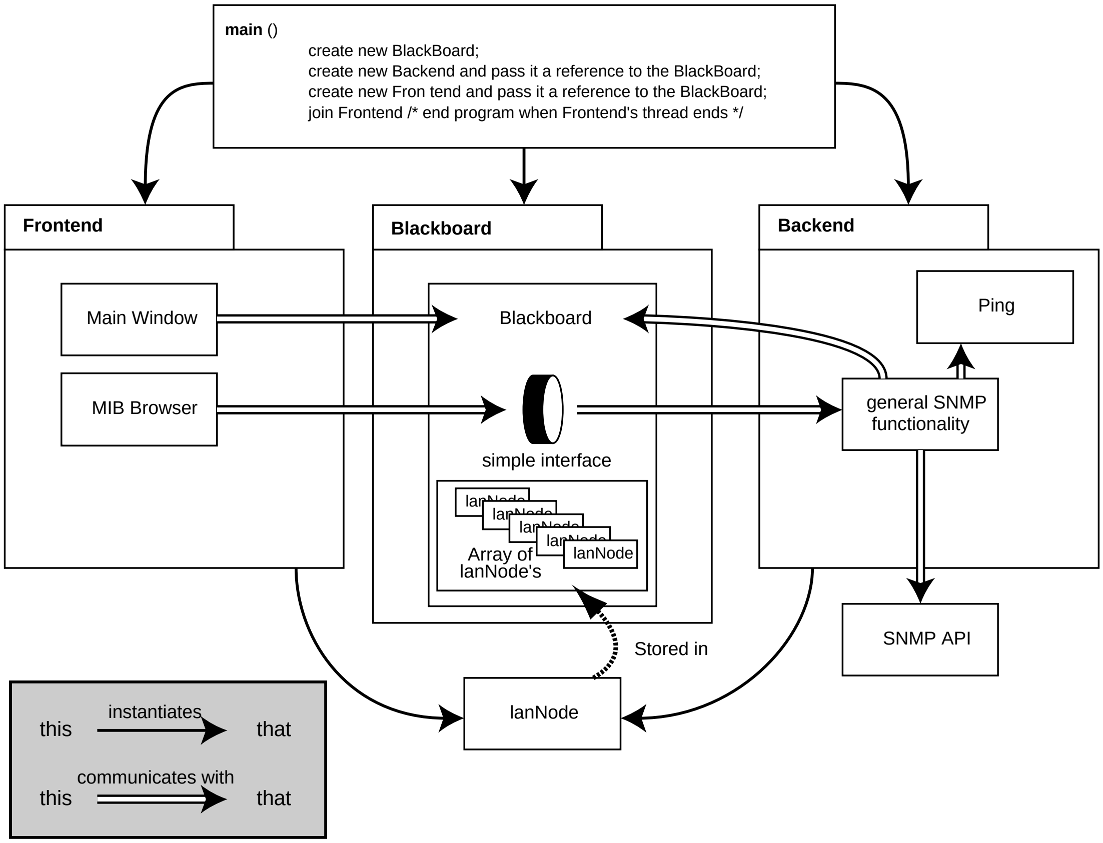
OID search attribute. If found, the corresponding OID is extracted from the XML file, an SNMP “get” is done on that value, and the user receives the result.

Interface with End-User

Simplicity is the main focus in the design of the user interface. The major function of the program is notifying the end-user when a computer has gone down, so some sort of audio cue is used as well as some sort of “eye-catching” visual alert. Also, the user needs to have a simple interface for adding nodes. With these two features, the main requirements are met. Anything above these two features are “extras” for more advanced users.

There are two main user-level modes available at run-time: Active, and passive. In the active mode, the program is constantly updating a display showing a simple representation of the network and highlighting machines that are not functioning properly. In this mode, the user can zoom in on a node and obtain a more detailed view. In the second mode, passive, the user can specifically request certain OID’s from a particular machine on the network, but no run-time probing is done.

Overview Diagram of NAPS



```
main ()
            create new BlackBoard;
            create new Backend and pass it a reference to the BlackBoard;
            create new Fron tend and pass it a reference to the BlackBoard;
            join Frontend /* end program when Frontend's thread ends */
```

Figure 2 NAPS Overview. This diagram shows an abstract overview of the interfaces between parts of NAPS, how program flow is divided through threads between the frontend and the backend, and what sections create instances of other classes.

III. Implementation

Frontend

As discussed in the high level design, the frontend's main task is to interface with the user, and the black board. The two main implementation designs to consider were the grid system and the linked list used. The grid system is the basis of displaying all the nodes that are being monitored, and the link list is how each item in the grid is being identified.

Grid System

The grid system is the foundation behind the frontend of NAPS. This system relies on dynamic, mathematical computations in which it will produce a graphical layout for nodes the user is monitoring. For simplicity of this algorithm, each node in this layout is represented by an image of fixed size.

As seen in Figure 3, each image is calculated dynamically according to the user's screen dimensions. Upon initialization of the application, the user's screen width and height are stored as INT values. From this, the width of the image is taken to be 10% of the user's screen width, and the height of the image is taken to be 86% of the image's width. Now that the image's dimensions are fixed, the grid system can place emphasis on arranging these images.

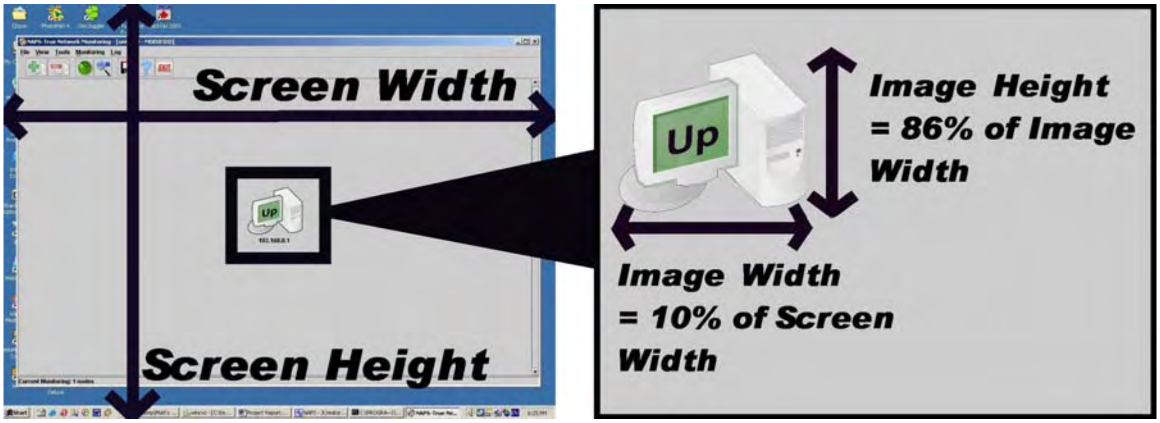

Screen Width
Screen Height
Up
Image Height = 86% of Image Width
Image Width = 10% of Screen Width

Figure 3 Screen Layout. General information about screen width, screen height, image width and image height

The grid system possesses two strategies for node placement. The first strategy is used if the number of nodes in the mapping is small, and if there is enough screen real estate. The second strategy is used if the first policy's requirements cannot be satisfied.

As seen from Figure 4, the grid system's first strategy is to place the images in a square fashion. This is accomplished by computing the square root of the number of nodes in the above mappings.

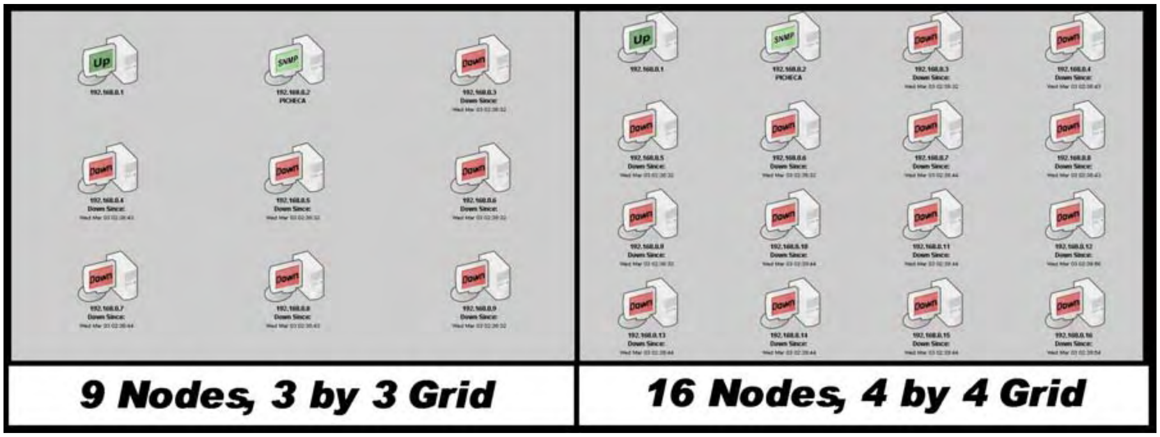


Figure 4 Sample Grids. On the left we show a 3 by 3 grid and on the right a 4x4. The above illustrates the square matrix case.

Although Figure 4 exemplifies mappings in which the number of nodes are perfect squares, Figure 5 demonstrates that the grid system will also function for mappings in which the number of nodes are not perfect squares. If the grid system cannot place the images in square fashion, it will position the images such that the number of rows and columns in the grid differ by only a small amount.

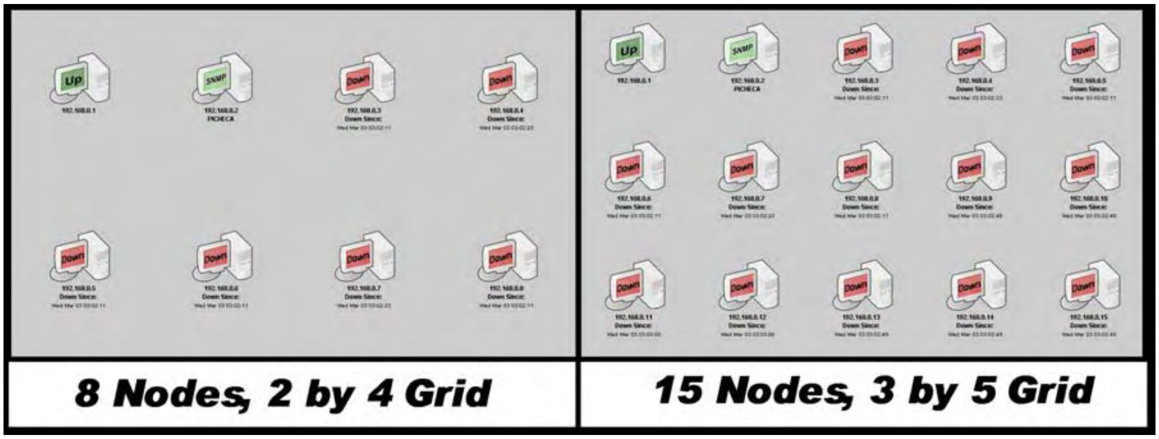


Figure 5 Sample Grids. On the left a 2x4 grid, and on the left a 3x5 grid. This illustrates that we also handle non-square matrices to efficiently handle screen space.

The pseudo code for the first strategy can be seen here:

```
// variables in double precision
double dblNumberOfRows, dblRemainder;

// variables represented as integers
int intNumberOfRows, intNumberOfColumns;
dblNumberOfRows = SquareRoot(Number of Nodes in Mapping);

// round down to produce an integer value
intNumberOfRows = Floor(dblNumberOfRows);

// obtain the value after the decimal place in dblNumberOfRows
dblRemainder = dblNumberOfRows – intNumberOfRows;

/* case where the number of columns needs to be 2 more than
 * the number of rows */
if (dblRemainder >= 0.5) then
        intNumberOfColumns = intNumberOfRows + 2;

// case where the number of columns will equal the number of rows
else if (dblRemainder = 0.0) then
        intNumberOfColumns = intNumberOfRows;
// case where the number of columns needs to be 1 more than the number
// of rows
else
        intNumberOfColumns = intNumberOfRows + 1;
endif
```

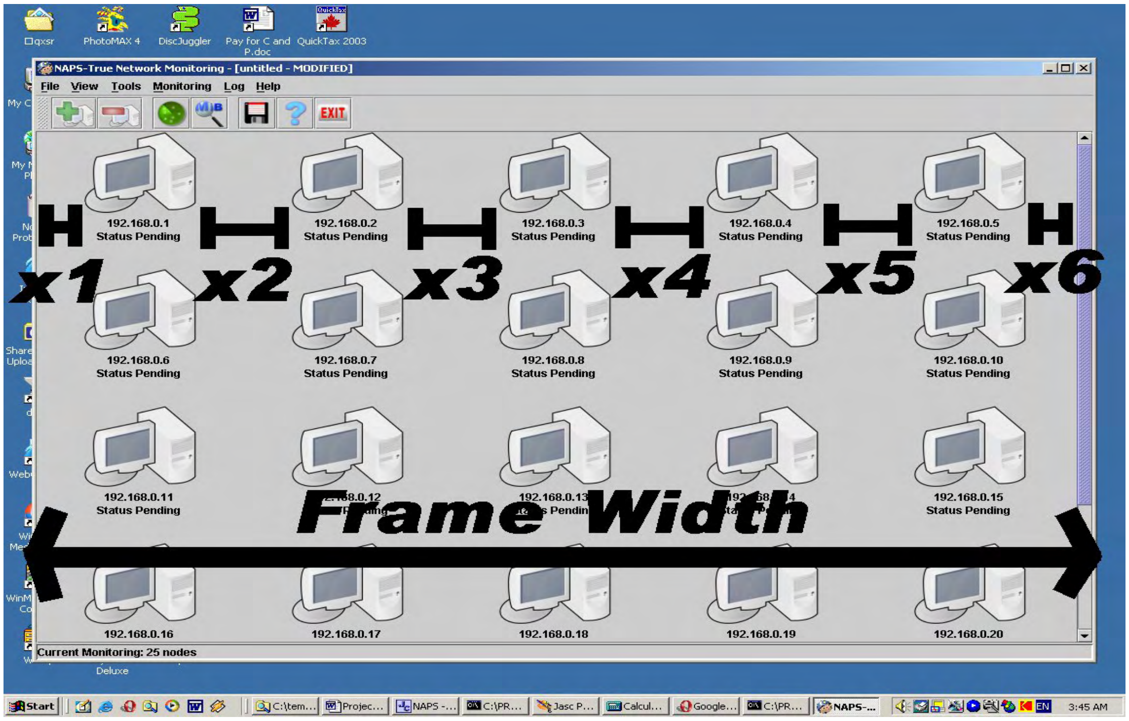

NAPS-True Network Monitoring - [untitled - MODIFIED]
File View Tools Monitoring Log Help
EXIT
192.168.0.1 Status Pending
192.168.0.2 Status Pending
192.168.0.3 Status Pending
192.168.0.4 Status Pending
192.168.0.5 Status Pending
x1
x2
x3
x4
x5
x6
192.168.0.6 Status Pending
192.168.0.7 Status Pending
192.168.0.8 Status Pending
192.168.0.9 Status Pending
192.168.0.10 Status Pending
192.168.0.11 Status Pending
192.168.0.12
192.168.0.13
192.168.0.14
192.168.0.15 Status Pending
Frame Width
192.168.0.16
192.168.0.17
192.168.0.18
192.168.0.19
192.168.0.20
Current Monitoring: 25 nodes
Start
3:45 AM

Figure 6 Phase 2 of the Grid System. This illustrates that a certain fix spacing ‘x’ between each element must exist or else a re-calculated area will be determined using a scrollpane.

The second strategy of the grid system can be further subdivided into two sub-phases. The first sub-phase is to verify if the first strategy displays the images in an appealing fashion. This can be seen through the following pseudo code:

```
int sum = 0;
// sum all the elements $x_1$ to $x_n$ (as seen in Figure 6)
for i = 1 to n
    sum = sum + x(i);
end for
// compare if spacing between images is appealing
if (sum < 3 * image width) then
    first strategy fails
endif
```

If the first phase indicates that the spacing between the images (as seen in Figure 6) is non-appealing, then the second sub-phase will be executed. The idea behind this sub-phase is simply to create appealing spacing between the images. This is accomplished by the following pseudo code:

```
int intNumberOfColumns, intNumberOfRows, intIdealSpace;
```

```
// set the desired image space to be the Frame Width (as seen in Figure 6)
// minus the ideal spacing between the images
intIdealSpace = Frame Width – (3 * Image Width);
// figure out how many columns can take place in this desired image space
intNumberOfColumns =  intIdealSpace / Image Width;
// now set the number of rows such that all the images
// will be contained in the grid
intNumberOfRows = Round Up (Number Of Nodes / intNumberOfColumns);
```

Finally, the scroll bar on the right hand side of the application is now enabled so the user may scroll up and down to view all the images.

ADT's used in Frontend

Due to the high number of possible end-user request operations that can occur, a well thought out abstract data type (ADT) needed to be implemented. The FENode object was created in order to deal with this high level of user-request. There are no real operations for modifying the data in the FENode class; it is just used as a storage medium. Some of this information includes: the corresponding LanNode, JLabel, hostname and IP address (in String format) for a particular node.

For the frontend to interact with each of the nodes in it's mapping, a second class is built to link the individuals FENode objects. This class is based on the fundamental structure called a linked list, which is referred to as a LinkedFENodeList. The difference between a standard linked list and our implementation is that the LinkedFENodeList places the nodes in sorted order, according to their IP addresses. This is accomplished through a method called: `public int compare(int[] insertingIP, int[] comparingIP)`. This takes as its arguments two IP addresses represented in an INT array with four cells in each array. Each cell represents an 8-bit INT value, which, when combined with the other cells in the array, forms the 32-bit IP address.

The method called compare performs a comparison on each 8-bit INT value (in sequential order) until either a difference is found between the two INT values, or the end of the array is reached. If all four 8-bit INT values are equal between the two arrays, -1 is returned by compare indicating that these two IP addresses are equal in value. Else, 1 is returned if argument insertingIP is less in value than comparingIP, and 0 is returned for otherwise.

For the `compare()` method to function properly, a second method is needed to transform the IP address in String value to a four cell INT array, which is termed: `public int[] StringToIntIp(String sInputIP)`. This method simply searches through the string sInputIP for periods, breaking the 32-bit IP address into four 8-bit INT values. If for any reason an error occurs, -1 is returned in each of the cells of the INT array.

The two main reasons for the frontend to be designed around this dynamic linked list are: the ease of future development and for an increase in performance. For instance, if a designer wishes to add more information to be stored for the frontend, all one is required to modify are the FENode and LinkedFENodeList class. This is a clear case where object programming is used to our advantage. We clearly separate frontend GUI aspects from our data making it easier to modify content being stored. Therefore, is it becomes necessary to add certain information you only need to modify the FENode object without having to worry about the rest of the frontend. However, if for example arrays are chosen as the storage medium, bounds are modified for add-ons to take place which can be tiresome and confusing. A second reason why the sorted linked list design is chosen is due to the time complexity, in other words performance. For the purposes of retrieving and storing data, the LinkedFENodeList methods are all of the order O(n) in the worst case. However, for the average case, the time complexity is much more efficient. For example, when storing a new FENode using the LinkedFENodeList class, the add method compares the inserted IP address of the node against the IP address for the nodes already in the list (as discussed before). If the user tries to insert a node which already exists, the add method searches through the list, find the already existing node, and cease activity because there is no reason to continue searching for an insertion point in the list.

Blackboard

The blackboard package can be considered the core of NAPS in that it is the central structure through which all portions of NAPS communicate. The package also houses the storage method by which information is maintained and accessed. From the beginning it was known that the NAPS application would be making use of threading so the package was also designed to be thread-safe (ie information may be altered by any thread without worrying about concurrency).

Please note that what follows is a detailed description of implementation of the BlackBoard, but most of these details can also be found through the HTML pages generated by JavaDoc which can be found on the program CD.

Storage Method

The blackboard package is divided into two major classes which store information about the network at different levels. The first class is the LanNode which stores information about a single node on the network. The second class is the BlackBoard which stores the LanNodes in a collection as well as keeping track of a few miscellaneous global variables.

In order to simplify updating information concurrently the LanNode is constructed as a collection of references and holds very little information directly. That way, if a copy of a particular LanNode is made, then both LanNode's point to exactly the same information and writing to one will reflect a change in the other. By this method, the frontend sees changes in information made by the backend instantaneously.

In order to allow all data in the LanNode to be changeable and changes reflected in all copies of a particular node, all data types in the LanNode need to be mutable. In most cases, the types of data that need to be stored are changeable objects, but in three cases alternate objects needed to be designed. The three cases were with Boolean values, Strings, and the InetAddress (which holds an IPv4 or IPv6 address and a hostname).

The Boolean values are a primitive type in Java which means they are always passed as values and are not treated as objects. If one were to copy an object with a Boolean value there would be no way to get both objects to point to the same Boolean value because you cannot have a reference to a primitive type. To get around this, a wrapper object is created as a nested class called a MutableBoolean. The MutableBoolean is a simple object which simply holds one Boolean value and has methods for getting and setting that value. This way, copies of a particular LanNode can point to the same MutableBoolean which then holds a single Boolean value. Also, the MutableBoolean is made as close as possible to Java's own Boolean wrapper class except that it is able to change its value.

Strings and InetAddresses are dealt with in exactly the same way as the Boolean and so that is why there are MutableString and MutableInetAddress nested classes in the LanNode.

One last major datatype used are the two mappings. There are two HashMaps in every LanNode for the backend to store OID-value pairs and the frontend to store whatever information is useful for it. The backend's mapping is chosen so there is no limit to how many pieces of SNMP information can be obtained (Java's HashMap will dynamically expand if the load factor is exceeded). The frontend has its own mapping so it stores information that was not thought about at the onset of the LanNode's design (i.e. easy extensibility for the frontend). However, all the information in the frontend's mapping is not saved when writing to a file as some of the objects stored in this mapping are Swing objects which are not Serializable.

One difficulty that was faced is how to tell if a particular node has been deleted. The only way to accomplish this is be to have a MutableBoolean flag which indicates when a particular node has been deleted. To improve efficiency, most methods in the LanNode are also made to throw a LanNodeIsDeleted exception if another object tries to update its information when the node has already been deleted. This comes in handy for both the frontend and the backend so that if any part of the program is working on a particular node when another part deletes it, it will immediately stop updating it and move on to something else.

The LanNodes also implement the Serializable interface so it is relatively easy to store and read in the information to and from a file.

The Observable interface is implemented by the LanNode so the frontend does not need to implement its own method of monitoring changes in the blackboard. Basically, whenever one of the LanNodes is altered the `update()` method in the frontend is called to update the display (the node changed is also passed to the update method as an argument). This makes use of Java's event broadcast/listener architecture.

The BlackBoard, as mentioned before, stores all the LanNodes in one collection. To accomplish this, the BlackBoard implements the Set architecture which is part of the Java Collection classes. The Set interface is chosen so there are no limitations on what can be stored in the BlackBoard. The most natural way to identify a particular LanNode is through its InetAddress but if this is the look-up key used to find a particular node in the BlackBoard then that restricts the BlackBoard to only having one node per IP address. Originally the BlackBoard was designed this way so a user could add multiple hostnames regardless of whether or not they pointed to the same IP address. For example, if "a.com" and "b.com" were both hostnames for the same machine, you could add both as separate nodes into NAPS. The reason for doing this is that most web servers allow for multiple sites to be hosted on a single machine and are only distinguished by the hostname used to access them. However, not having a specific look-up key causes access to the BlackBoard to be much slower since the only method of searching is to iterate over the set (ie sequentially searching an unordered set).

Concurrency

To ensure that there are no race conditions with updating a LanNode's information, a locking mechanism is implemented to make sure that only one writable copy of a LanNode exists at any given time. This way, there is a read-by-many/write-by-one type of design which means that if an object is writing to the LanNode it can be confident that it is the only object writing to it. However, this is used as more of a design constraint so that the blackboard package can ensure that neither the frontend nor the backend check out a node for writing more than once. If an object tries to check-out a node more than once than an exception is thrown.

Unlike C, since Java is implemented on a Virtual Machine then it is difficult to determine if any command is actually atomic. Because of this, it is nearly impossible to implement your own mutual exclusion locking system using just Java code. Thankfully, Java comes with a method for locking objects and the lock is automatically enforced, so a block of code can not simply ignore the mutex as is possible in C. The method uses the command synchronized to create a block of commands where only one thread may access those commands at a time (other threads wait automatically for their turn).

In most cases, the synchronized keyword is used as a method descriptor which is equivalent to putting the whole contents in a `synchronized(this) {}` code block. This means that no two threads can make changes to a LanNode at any given time. However, this is also already enforced with the locking mechanism designed early in the design of the LanNode which uses a simple Boolean flag.

Backend

The backend must interface with our blackboard, but must also go out and probe for remote nodes verifying their status. The key implementation issues are how we manipulated our threads, listening for SNMP traps, and manipulating individuals OIDs.

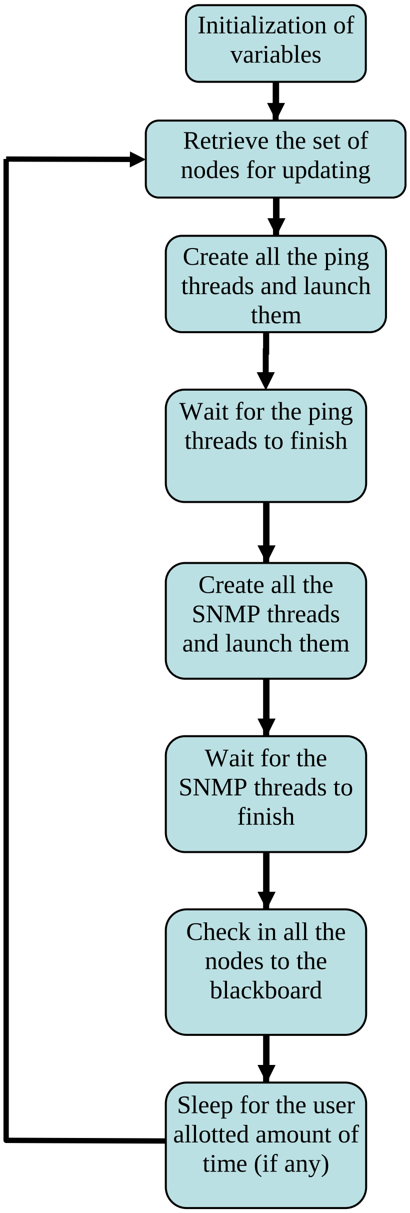

Initialization of variables
Retrieve the set of nodes for updating
Create all the ping threads and launch them
Wait for the ping threads to finish
Create all the SNMP threads and launch them
Wait for the SNMP threads to finish
Check in all the nodes to the blackboard
Sleep for the user allotted amount of time (if any)

Figure 7 Backend Flow. General flow of the backend thread class.

Threaded Backend

Java's support for threads is build around two classes within the java.lang package: Thread and ThreadGroup. Our backend method is designed as a child to the Thread class by extending it. The backend acts as the dispatcher that creates individual thread instances for each node that is extracted out of the blackboard. It also controls how many nodes are taken out of the blackboard and spawned. The user, in the frontend is capable of controlling how many threads are created at once. The backend keeps track of every thread it creates by keeping an array of the currently spawned threads. Figure 7 illustrates how the flow of the backend works.

Example Code for the ping and SNMP threads:

```
      pingArray = new
pingThread[NUM_THREADS];
      snmpArray = new
snmpThread[NUM_THREADS];
```

NUM_THREADS represents the total number of threads set by the user in the frontend. The ordering of which nodes get updated is dependent on the order they get retrieved by in the blackboard. A simple iterative for loop gets up to the maximum nodes allowed if the number of nodes is less then the maximum number of threads set by the user then all the nodes are retrieved.

Example Code showing NAPS queuing system:

```
//Get up until either
//all nodes are checked out
//or max number of threads reached
while ((tempNode != null) && (numNodes <
NUM_THREADS)) {
    tempNode = BBMethods.getForUpdate();
    if (tempNode != null) {
        nodeArray[numNodes] = tempNode;
        numNodes++;
    }
}
```

Where *tempNode* stores the current node being retrieved and *nodeArray* stores the current set of nodes retrieved for updating.

The creation of the threads is done by creating an instance for every thread stored in the *nodeArray*. This creates exactly the number of threads needed to update the current number of nodes we checked out. After the threads are all created and spawned off it waits for them to complete by joining them into the main thread (This will destroy the thread instance).

Example of creating the ping threads:

```
// (the SNMP threads are done in the same fashion)
//Start pinging all the nodes we checked out note:
//(MAX NODES CHECKED OUT IS = NUM_THREADS)
for(tempCounter = 0; tempCounter < numNodes;  tempCounter++) {
    pingArray[tempCounter] = new
pingThread(nodeArray[tempCounter]);
}

//Wait for all the threads to finish
try {
    for(tempCounter=0; tempCounter < numNodes; tempCounter++) {
        pingArray[tempCounter].join();
    }
}
```

The updating of the information when interfacing back to the blackboard is done in the ping and SNMP thread classes. So once the threads are terminated the nodes are checked back inside the backend.

SNMP Trap Listening & Trap Processing

SNMP traps are sent by remote computers and the backend acts as a listener for these traps. The default port that the listener uses is port 169 this is the standard SNMP trap port. The trap server is run very simply by creating a trap listener from the SNMP library as follows:

```
try {
trapListenerInterface = new SNMPv1TrapListenerInterface();
trapListenerInterface.addTrapListener(this);
//Start listening for traps
trapListenerInterface.startReceiving();
}
```

The trap processing is done through a method that is automatically called from the trap listener server. Depending on whether the node exists in the mapping or not the trap message are processed and sent to the blackboard for user notification.

SNMP information retrieval & setting

Interfacing correctly with our external SNMP library is essential to our success. The backend is responsible for creating a communication interface with remote nodes. When attempting to get information from a remote agent, the returning value is the string representation of the corresponding OID. The setting of SNMP information is done in the same manner but except for asking for specific information, the function sets the specified information and depending if it is successful or not, returns true if successful and false otherwise.

Example of `SNMP interfacing code`

```
//Create the SNMP interface
SNMPv1CommunicationInterface comInterface =
new SNMPv1CommunicationInterface(version,
hostAddress, community);
//Get the value of the corresponding OID
value
SNMPVarBindList newVars =
comInterface.getMIBEntry(OID);
```

Threaded Pinging

Pinging is done by spawning an external process call to the local operating system’s ping executable. Since NAPS is platform independent, one needs to determine the current operating system, and call its ping accordingly. Once the ping is called the output is parsed to determine if the node exists on the network. The outputs for the three different platforms are distinct and were taken into account.

The ping thread relies on the following system calls to force the OS to spawn an external process.

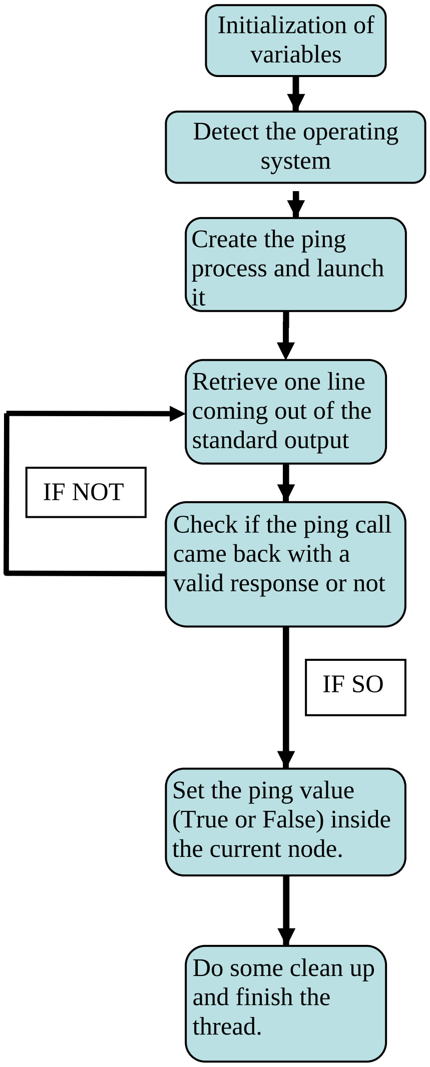

Initialization of variables
Detect the operating system
Create the ping process and launch it
Retrieve one line coming out of the standard output
IF NOT
Check if the ping call came back with a valid response or not
IF SO
Set the ping value (True or False) inside the current node.
Do some clean up and finish the thread.

Figure 8 Ping Thread Flow. This flow chart represents the life of the ping thread once spawned.

Example for Get the OS System type:

```
        String osType = System.getProperty("os.name").toLowerCase();

        //Launch the ping process
        helpProcess = Runtime.getRuntime().exec("ping -n 1 -w 1000 "+ ip);
```

The method in which the output of the ping process is read in is using an input stream reader attached to the process

Example of the input buffer used:

```
        progInput = new BufferedReader(new
        InputStreamReader(helpProcess.getInputStream()));
```

As shown at the left in Figure 8, the ping thread class reads in each line separately until either we can safely determine that no response is coming (i.e. the ping call failed) or if we can determine whether the ping call succeeded (i.e. the ping call returned).

The SNMP threads are created in a very similar way to the ping threads it; even uses the same node array to select which nodes are to be updated. The flow of the SNMP class is as follows in Figure 9.

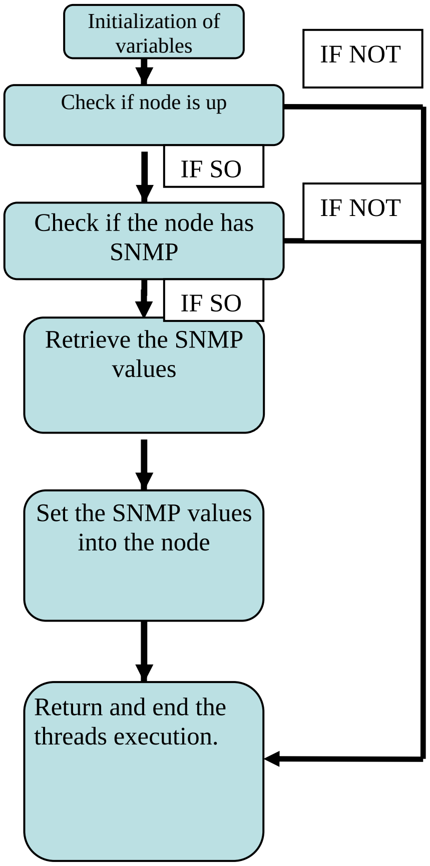

Initialization of variables
IF NOT
Check if node is up
IF SO
Check if the node has SNMP
IF NOT
IF SO
Retrieve the SNMP values
Set the SNMP values into the node
Return and end the threads execution.

Figure 9 SNMP Thread Flow. This flow chart represents the life of the SNMP thread once spawned.

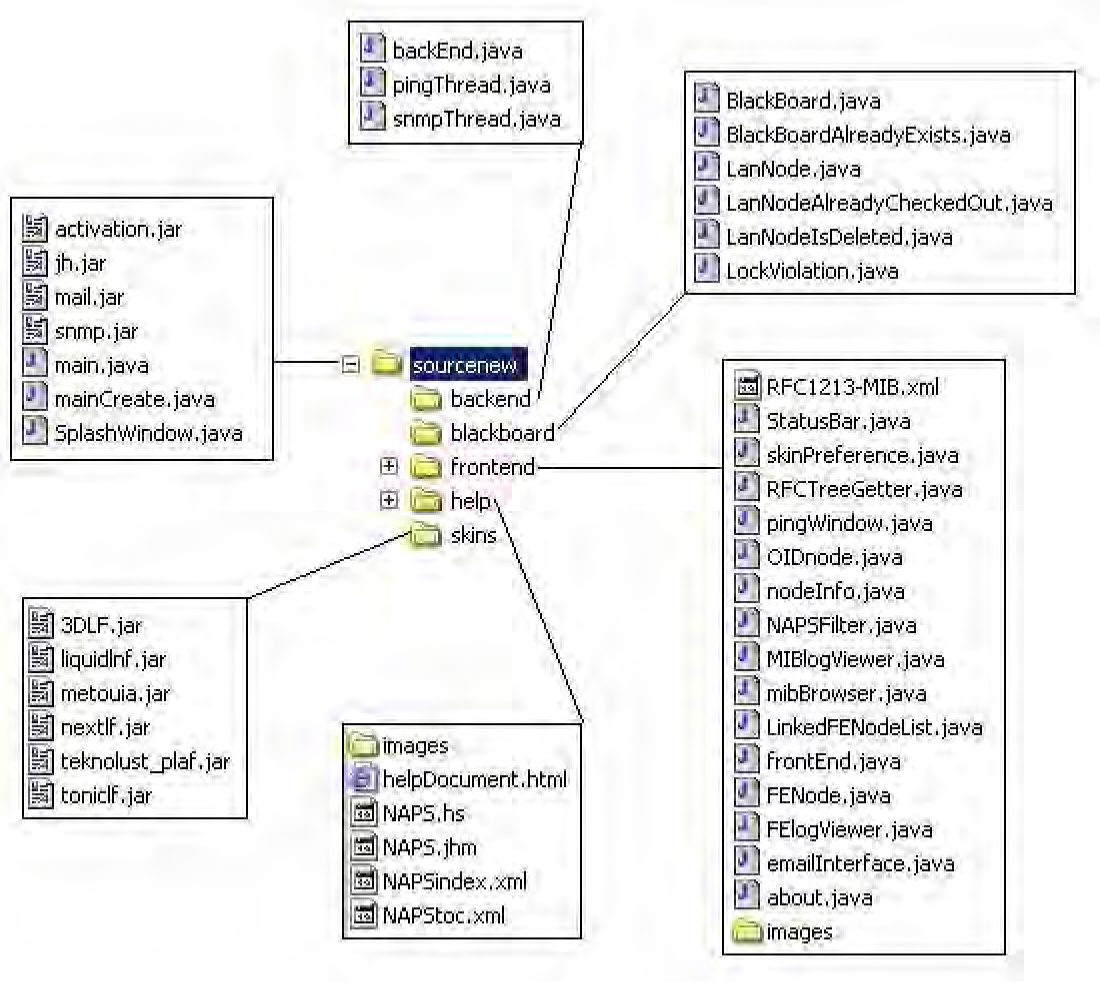
External SNMP Library

The external library is chosen for the SNMP interfacing is one that was made by Jonathan Sevy. The main reason it was chosen is because it was written in pure Java; it had all the features we needed; it was well documented and easy to use, and most importantly it was open sourced and free.

The SNMP library that was used in NAPS can be downloaded from: http://edge.mcs.drexel.edu/GICL/people/sevy/snmp/snmp.html

File Listing

With the key algorithms and code discussed we wanted to just show the basic hierarchical breakdown of our Java files. Figure 10 shows exactly how we divided the project to be more efficient. Our root directory, called sourcenew, contains the main.java file which is the execution point of our program. The complete SNMP (snmp.jar), JavaHelp (jh.jar, activation.jar), and Mail (mail.jar) libraries that we used are also located in sourcenew. The skins subdirectory contains the libraries for allowing the user to change the look and feel of the programs interface. The help menu simply contains the helpset file (NAPS.hs), and the corresponding XML and HTML files that comply with JavaHelp (version 2.0). The other directories: frontend, backend, and blackboard all contain the core java files that were used. For detailed explanation of the available functions for each Java file please view the JavaDocs. JavaDocs can be obtained either by the distribution CD or online at http://www.creativestudent.com/naps/ .



```
sourcenew
    backend
    blackboard
    frontend
    help
    skins
```

sourcenew:
activation.jar
jh.jar
mail.jar
snmp.jar
main.java
mainCreate.java
SplashWindow.java

backend:
backEnd.java
pingThread.java
snmpThread.java

blackboard:
BlackBoard.java
BlackBoardAlreadyExists.java
LanNode.java
LanNodeAlreadyCheckedOut.java
LanNodeIsDeleted.java
LockViolation.java

frontend:
RFC1213-MIB.xml
StatusBar.java
skinPreference.java
RFCTreeGetter.java
pingWindow.java
OIDnode.java
nodeInfo.java
NAPSFilter.java
MIBlogViewer.java
mibBrowser.java
LinkedFENodeList.java
frontEnd.java
FENode.java
FElogViewer.java
emailInterface.java
about.java
images

help:
images
helpDocument.html
NAPS.hs
NAPS.jhm
NAPSindex.xml
NAPStoc.xml

skins:
3DLF.jar
liquidlnf.jar
metouia.jar
nextlf.jar
teknolust_plaf.jar
toniclf.jar

Figure 10 File Listing Breakdown

IV. Verification

Overview of Testing

Our test plans were divided into four phases. The following paragraphs describe each. It is important to note that all were required in order to say we tested the software from bottom-up and top-down. Phases 1 and 4 were the most critical and provided the most feedback. Phase 2 and 3 guarantees that our modules are reliable and robust. Many features such as:

- UI design considerations
- Blackboard Concurrency
- Network Congestion
- Memory Leak tests 24/48/76 Hour Test
- Multiple OS testing (Windows/Unix/Linux)
- Stress test with very large number of calls/threads

were tested very closely within the following test plans.

Phase 1 Testing – Frontend and UI considerations

Ethnography is part of an anthropological approach seeking to understand and describe the points of view of members of a particular culture by conducting intensive field works, such as careful observations and in-depth interviews to describe, in detail, the points of view of the informants. Information will be collected to be able to modify and optimize our computer human interface (CHI). The factors that we closely monitored are the following:

- **Time to Learn**

  The length of time it takes for the users to learn how to do a task
- **Speed of Performance**

  The length of time it takes for users to carry out a benchmark task
- **Rate of Errors made by users**

  The number of errors made by a user who is carrying out a task

  The number of types of errors made by users
- **Retention of CHI Operations over time**

  Test the users' ability to remember how to complete certain tasks after a period of time.
- **Subjective Satisfaction**

  Which aspects of the CHI were best appreciated by the users?

  Overall rating

The above five criteria are often the key factors that can distinguish a “good” CHI versus a “bad” CHI. We conducted a usability study with eight individuals with various backgrounds to obtain results. The users vary from being network administrators to novice computer users. This allows us to get a feel for exactly how robust our UI will be. If major issues are discovered, or if certain frontend aspects need modifications, Matthew Picheca is responsible for fixing and re-testing the package.

Phase 2 Testing - Blackboard Testing

The blackboard class is a set that holds all our data; what we refer to as LanNodes. Every LanNode that is created corresponds to a device on the network. Two main issues can arise in the blackboard: reaching maximum capacity and problems arising due to concurrency. Since the blackboard works similar to a library, where the frontend and backend can check out LanNodes, it must be able to lock certain tasks from being done at the same time to prevent erroneous data. The other main issue that must be documented is to what degree our blackboard is reliable? This is where JUnit (version 3.8.1) comes into play. This software package allows us to create individual test suites and test them with JUnit’s built in assert checking. The output is displayed graphically and with a full progress indicator. If an error is found, JUnit will display what failed and at what location. It is then up to the programmer to look closer into the problem. It also lets us know how well it works under repeated use, by telling us how much time it takes to complete a task. The JUnit test module is provided in Appendix D. If any failures occur, the issue will be addressed to Tim Tisdall, who is the designer of the blackboard.

Phase 3 Testing - Backend Testing

Unit testing was also done here to exhaust all possible results that the “ping” program can return on all operating systems that we will support. These include: Microsoft Windows 98/2000/XP, Red hat Linux 9.0, and Solaris 8.0. Those are the operating systems we will perform our test on and guarantee correct results for. Other analysis that underwent for this unit of the testing were calculating delay times in the backend, as well as comparing iterative versus threaded performance. Paul is responsible for the design of this test module and it is included in appendix E.

Phase 4 Testing - Regression, Integration, and Stress Testing

The final phase of our testing was the overall integration test. This test allowed us to see the overall program’s robustness when it was all put together. We combined regression, integration, and stress testing into one phase. Regression testing was completed in order to review our previous bug fixes from phases 1, 2, and 3. Integration testing was performed since all three main components, as well as external XML files and libraries were called upon. Finally, by running the code for prolonged periods of time in various real-time environments, it satisfied our stress testing requirements. We ran the program on Red hat Linux 9.0, Solaris 8.0, and Windows 2000 for periods of 24, 48, and 76 hours and logged CPU performance results. We used a combination of tools to check our CPU utilization factors over the period of time the program would run for. Tools such as FreeMeter version 2.7.7, Norton SystemWorks 2004, and various build-in Windows tools were all included in this phase. We were mainly concerned with memory leaks, or over utilization of CPU resources, over long periods of time.

The cost of constant polling and network traffic was very important to us, as well as any network administrator. In order to show that using our tool will not result in any congestion issues, we tested our software using a network traffic sniffer. Sniffer Pro version 4.50 by Network Associates is what we installed to monitor network activity in real time. This software is designed to collect detailed utilization and error statistics for individual stations, as well as whole portions of networks. By changing the number of threads we spawned to the maximum and reducing our ping intervals, we measured the effect on network traffic.

Phase 1 Testing Results

Located in appendix C at the end of this document you can see the general structure we followed during our interviews. The questions gave us feedback on all five key aspects, which allowed us to conclude the overall success of our CHI. The purpose of conducting this survey and collecting a set of beta testers was twofold. One, we wanted to make sure that no one else could crash the program. The second reason was to get feedback on our user-interface. By completing this, we obtained information telling us where we fell short of proper design and possible minor design choices. In turn, we were able to fix these minor bugs before its official release date. The following list provides the constructive criticism on aspects that we fell short on for our user interface design:

- Splash Screen: Let the user know how to close the splash screen when he loads the about window.
- The majority of our beginner users wanted help, something to get them kick started. Hence a “Get Started Wizard” should have been implemented.
- Users with low levels of networking knowledge were not sure of the format of an IP Address. Popup help should be implemented or a re-designed UI.
- When double clicking on a node, if certain values aren’t listed (for example sysLocation) then prompt the user to set the values.
- Allow for more skins to be used, or even the ability to use user-defined skins.

- Users wanted the ability to customize icons, or to be given a choice as to what icons to have loaded with a corresponding computer.
- MIB Browser had some compatibility issues that gave off warnings on the console.
- No one knew till reading the user's manual how to add a range.

Looking at the first criteria, we wanted to test "Time to Learn"; we discovered that with the number of shortcuts and menus we give a user, the time it took for them to complete common task, such as adding and deleting nodes, was small enough not to quantify. We were quickly able to conclude our success in this category.

The second criteria varied significantly depending on the user's computer background. The task that we gave them was to obtain a specific OID on the local host (that already had SNMP enabled). The timed results from eight testers are as follows:

Table 3 Time to Learn Results. This information was collected by timing testers on completing certain tasks.

Tester/Gender	Time till Successfully Executed
Tester 1 (Male)	18.6 Seconds
Tester 2 (Male)	16.4 Seconds
Tester 3 (Female)	29.3 Seconds
Tester 4 (Male)	34.2 Seconds
Tester 5 (Male)	12.5 Seconds
Tester 6 (Female)	15.1 Seconds
Tester 7 (Male)	17.6 Seconds
Tester 8 (Female)	19.8 Seconds

What we concluded with the above table is that overall, the time it took to complete this task was not overwhelming. Since it is one of the longest tasks to do in NAPS, with the highest level of difficulty, we found this to be a true test as to whether or not people were able to find their way into the program.

The majority of the interview was spent watching the user click around with the software. By doing this, we were able to record certain bottleneck points that would frustrate the user. For example:

- Adding an IP address (i.e. bogus inputs were added)
- Attempting to set OID values that only had read permissions attached to it.

Finally, we also wanted to know if people remembered how to use the product over time, and their overall satisfaction. We tested this by asking our testers 24 hours and 48 hours after our interview, certain specific questions on how to complete a task. If the user was verbally able to remember and explain what to do, then we had succeeded in our design. The table with our results can be found below.

Table 4 Memory Retention of User Interface. We randomly followed up with the testers to see if they remembered how to complete a certain event. If they did the table shows a ‘success’.

Tester/Gender	24 Hours Later	48 Hours Later
Tester 1 (Male)	Success	Fail
Tester 2 (Male)	Success	Success
Tester 3 (Female)	Success	Success
Tester 4 (Male)	Success	Fail
Tester 5 (Male)	Success	Success
Tester 6 (Female)	Success	Success
Tester 7 (Male)	Success	Success
Tester 8 (Female)	Success	Success

Our overall rating from our testers shows a promising outlook for the program. 8.5/10 is what the average overall satisfaction was.

Phase 2 Testing Results

The test module can be broken down into the following eight sections:

1. The first part of the testing does not test the blackboard at all; it just tests if all the correct settings and variables are set. This test should only fail if the programmer enters conflicting or invalid parameters inside the code.
2. The second part of the testing is when the blackboard is first created, the program checks if the blackboard is truly empty and if it exists. If this test failed, then there would be a problem with the creation of the blackboard.
3. The third part of the testing tests the blackboard when it is filled with a random number of nodes (less then or equal to half the max node limit). The reason a random number of nodes were added was because testing the blackboard using varying input data was a much more robust and true-to-life test. This test should fail if adding any of the nodes to the blackboard was unsuccessful, if the size reported by the blackboard was incorrect, or if changing the parameters of the newly added node caused it to fail unexpectedly (All available parameters will be modified and tested).
4. The fourth part of the testing tests the blackboard when it is completely filled (This level is determined inside of the testing program and can be changed easily). The blackboard is filled with newly created unique IP addresses and once all the

nodes are added, it is checked in the same manner as in the third part and failures are quickly caught and displayed.

5. The fifth parts of the testing tests the blackboards delete all method. This method should delete all the contents of the blackboard and nothing should remain. This will fail if the blackboard still has any elements inside of it.
6. Same as Third Part
7. Same as Fourth Part
8. The eight part of the testing tests the blackboard on how it stands up to a deletion of every node individually. If any of the nodes either don't exist or if the blackboards size varies from what is expected, then a failure is recorded.

The third to eighth part can be repeated any number of times depending on the settings inside the program code. This tests the program under a high amount of stress and repeating adding and deleting of nodes.

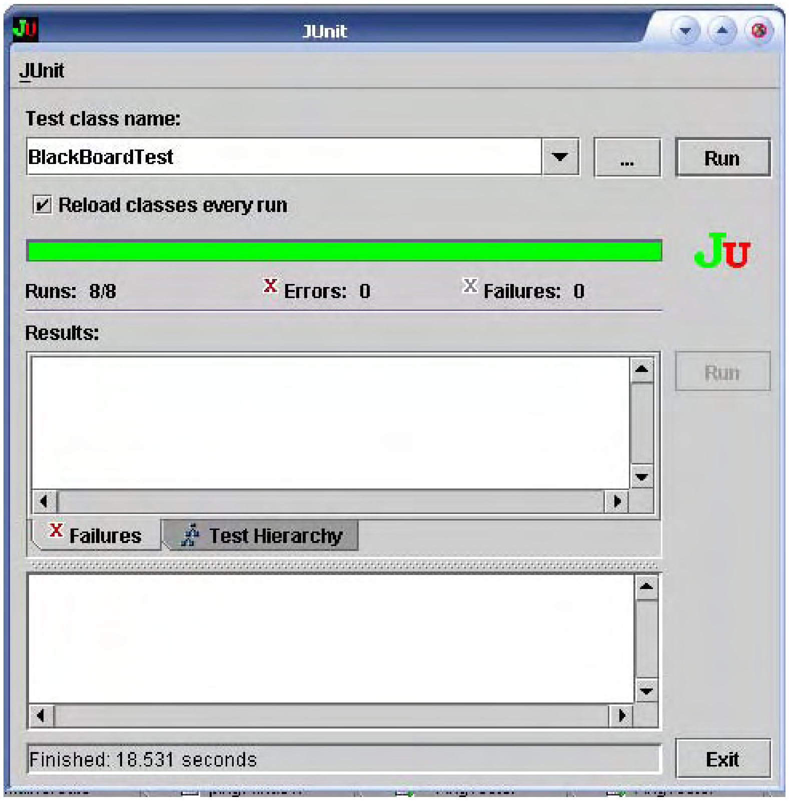
Results of testing

The results of the testing were quite promising and did not indicate any errors up to 5000 nodes.

Table 5 Timed Random Node Generator Results.
The amount of time it took for the simulator to generate different amount of nodes

Number of Nodes	Result	Time taken (seconds)
2	Pass	0.11
10	Pass	0.594
50	Pass	2.875
100	Pass	6.063
200	Pass	11.203
500	Pass	18.531
1000	Pass	61.031
2000	Pass	100.84
5000	Pass	375.156

Above 5000 nodes the random IP generator takes much too long to keep creating random IP addresses and thus only 5000 nodes were tested.



Test class name: BlackBoardTest

☑ Reload classes every run

Runs: 8/8 Errors: 0 Failures: 0

Finished: 18.531 seconds

Figure 11 JUnit UI for Blackboard. A sample output of 500 nodes being tested.

Example output:

```
Testing constraints....Done!
Adding 49 ip addresses!....Done!
Testing newly added nodes...Done!
Adding 451 ip addresses!....Done!
Testing newly added nodes...Done!
Testing quick delete of all nodes....Done!
Adding 79 ip addresses!....Done!
Testing newly added nodes...Done!
Adding 421 ip addresses!....Done!
Testing newly added nodes...Done!
Testing single delete of all nodes....Done!
```

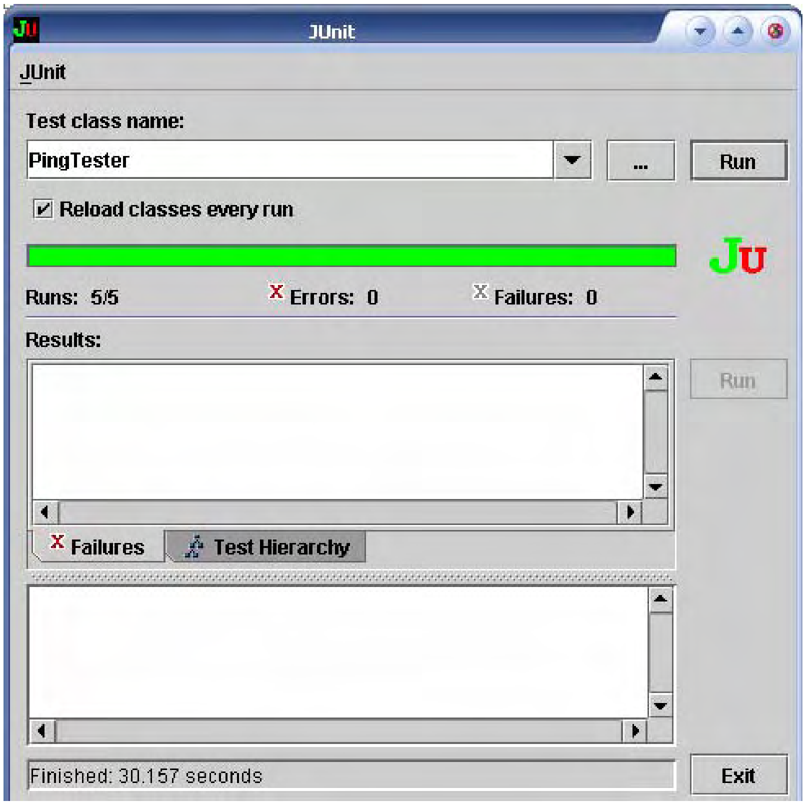
The testing of the blackboard went as expected. It was able to hold a large number of nodes without crashing or failing any of the rigorous tests. The tests were complete and tested almost every part of the blackboard.

Phase 3 Testing Results

Once again to gather proper results using JUnit, we divided our test module into five parts:

1. The first part of the testing does not test the pinger, it just tests if all the correct settings and variables are set. This test should only fail if the programmer enters conflicting or invalid parameters inside the code.
2. The second part of the test adds in 3 special nodes in the blackboard. The following are the nodes that were added:
   - 127.0.0.1 (Loop back node should always return true)
   - 0.2.3.4 (unreachable node should always return false)
   - 0.0.0.0 (invalid node should always return false)

   If any of these nodes cannot be added to the blackboard, a failure is reported
3. The third part of the testing adds the remaining deficit of nodes to the backend. Say the programmer specifies 3 nodes to be tested then the tester will add the 7 remaining random nodes to the blackboard giving a total of 10 nodes.
4. The fourth part tests the nodes using threads. It retrieves the nodes from the blackboard and pings them with the predetermined amount of threads. Any unusual behaviour in this method will be noted and displayed as an error. Also, the 3 special nodes noted above are also pinged and their results are compared to what is expected. The amount of time it took for the nodes to execute is also recorded and displayed. (This information was used to test the performance of the threaded pinging and iterative pinging).
5. The fifth part of the tests the iterative pinging. It checks out all the nodes from the blackboard pings each one, one at a time until all the nodes are finished. Any unusual behaviour in this method is recorded and displayed as an error. Also the 3 special nodes are tested as well and the results are compared to what will be expected. The amount of time it took for the nodes to execute is also recorded and displayed. (This information was used to test the performance of the threaded pinging and iterative pinging).



Test class name: PingTester

☑ Reload classes every run

Runs: 5/5 Errors: 0 Failures: 0

Finished: 30.157 seconds

Figure 12 JUnit UI for pinger. A sample output when testing 30 nodes.

Example output:

```
Adding special node 127.0.0.1
Adding special node 0.2.3.4
Adding special node 0.0.0.0
Adding 7 random ip addresses!....Done!
Testing Threaded pinging....Done!
It took approximately 21 seconds to ping 10 nodes using threads
Testing Iterative pinging...Done!
It took approximately 8 seconds to ping 10 nodes iteratively
```

As part of analysing this backend component, we wanted to quantify the amount of time it takes to checkout a number of nodes. This result is dependent on the number of seconds the user sets in the frontend and the amount of time it took to update the current number of nodes. Here is a pseudo code example of calculating the yield time:

```
Yield Time = User Set Time – Time Elapsed in updating
```

Actual code used to calculate and execute the yield time:

```
//Start the timer!
t0 = System.currentTimeMillis();

... (run all the updates)

//End the timer!
t1 = System.currentTimeMillis();

dt = t1 - t0;

if ((dt/1000.0) < PING_INTERVAL) {
    //Calculate sleep time
    sleepTime = (PING_INTERVAL * 1000) - dt;
    //yield processing to other threads for the given time
    try {Thread.currentThread().sleep(sleepTime);}
    catch(InterruptedException e) {/*Do Nothing*/}
}
```

As one might notice in the above code, if the updating takes longer then the user set ping interval, then no waiting happens and the updating is done in a continuous loop, but since the backend is a thread itself it automatically yields to the frontend and does allow other processes to execute.

Performance of Threaded Pinging vs. Iterative Pinging

The main reason that the pinging and SNMP calls are threaded is because of performance, efficiency, and responsiveness issues.

Performance can be measured by how long it took for “x” amount of ping calls to return using threads compared to the iterative way. Looking at the graph below, we can see there is a significant difference between the two versions of ping, even though they do the same thing, each in a different way.

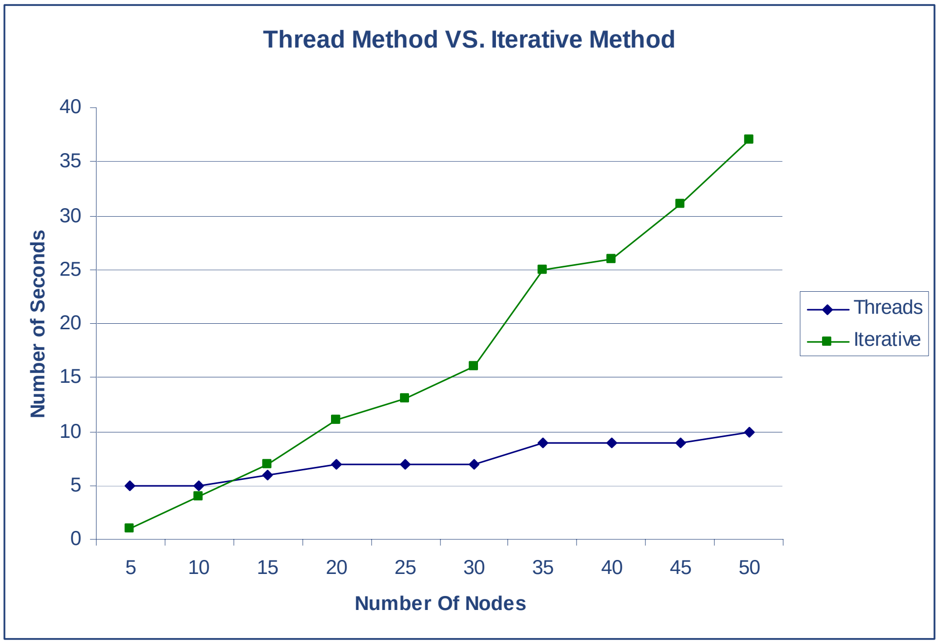


Figure 13 Threads vs. Iterative. This graph illustrates the efficiency of threads.

As we can see, with the above graph with a lower number of nodes, the iterative method is a better choice because of the overhead that is required with the thread method. As we increase the number of nodes, we can see that the iterative method has a sharp increase upwards while the thread method increases much slower. At approximately 11 nodes, the thread method overcomes its overhead and starts becoming faster then the iterative method. (Note: Even though the iterative method is faster at a lower number of nodes it does not allow of any other process to run thus responsiveness of the program suffers.)

Performance depending on the number of threads

The number of threads has a significant effect on the number of seconds it takes to ping “x” amount of nodes. The more threads that are dispatched the faster the nodes get updated. The following graph shows the result when 30 nodes were pinged.

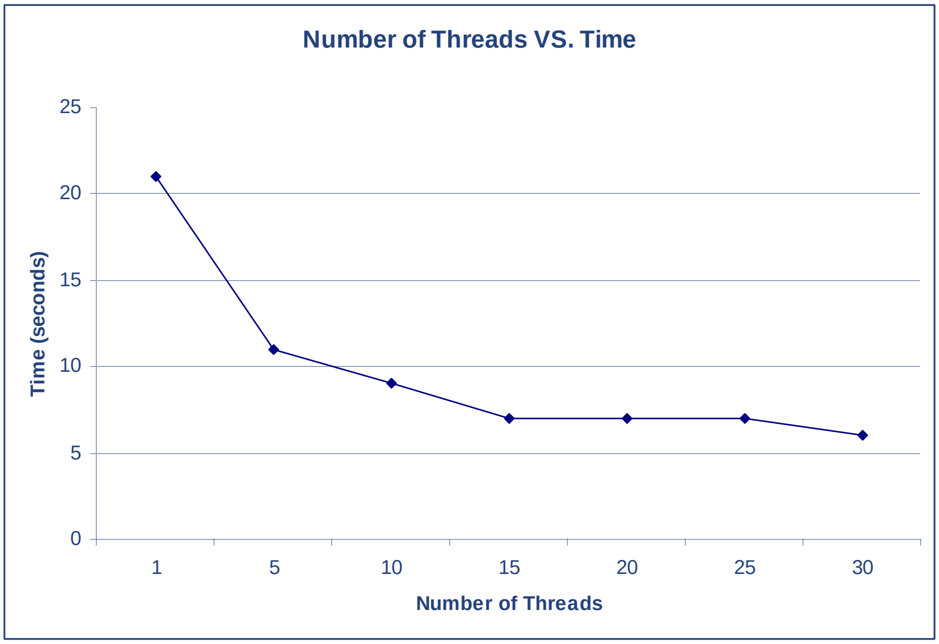


Figure 14 Number of threads vs. Time. This graph displays the link between the number of threads and amount of time it takes to complete their life cycle.

Clearly, as the number of threads increase the time in seconds to ping the 30 nodes decreases, but only to a certain point. As the number of threads approaches the number of nodes, the time it takes for the pinging to complete does not change.

In conclusion, the analysis for this phase was critical since it was the underlying mechanism that fed our frontend data. Although our implementation holds, relying on an outside program to probe the network is not the best approach. Many other alternatives can be studied in the future such as using Java Native Interface to create an ICMP library in C and interfacing it in the backend.

Phase 4 Testing Results

This was our most informative testing phase, based on all the discoveries we made during this time period. This phase allowed us to package everything together and see how it worked on randomly selected machines for stretched periods of time. We discovered initially that we had a large number of bugs to fix. The initial key problems arose when overlooking certain thread issues that were occurring. NAPS works by spawning ping threads, and these threads need to be carefully executed and terminated. If they are not terminated, the program would overflow the PC’s memory. We had forgotten to clear certain threads, rendering a computer to the ground after a period of time. Our next big discovery was made when monitoring large networks at the highest levels of network probing. This caused an exception, which created an avalanche effect of problems that varied from run to run. The exact exceptions that were occurring at random points after execution were the following:

```
IOException java.io.IOException: The specified procedure could
not be found
IOException java.io.IOException: Bad file descriptor
```

The above two bugs were amongst the hardest to understand and locate in the source. Paul was responsible for the fix on the above two propagating exceptions. Many hours were spent using Java's built in methods for dealing with memory leakages specifically (`java.lang.Runtime.getRuntime().freeMemory()`).

The other major revisions that were needed at this point in our testing were to deal with proper package distribution. Since we dealt with external libraries, file reading/writing, and image loading, we had to change the majority of our code that dealt with I/O to take into consideration that everything was zipped into a JAR file instead. So, to hardcode specific paths was creating problems when distributing our software. The solution involved using relative paths; in Java these are referred to as URLs.

While concurrently finding the major issues stated above, we were performing our real-time stress tests. Below is a chart of the results we obtained.

Table 6 Stress Test 200 Nodes. These were our results after two distinct rounds when monitoring 200 nodes in a real time environment.

(200 Nodes)	**Round 1**			**Round 2**		
	24 Hrs	**48Hrs**	**76Hrs**	**24Hrs**	**48Hrs**	**76Hrs**
Microsoft 2000 Pro.	Fail	Pass	Pass	Pass	Pass	Pass
Microsoft XP	Fail	Pass	Pass	Pass	Pass	Pass
Solaris 8.0	Fail	Pass	Pass	Pass	Pass	Pass
Red Hat Linux 9.0	Fail	Fail	Pass	Pass	Pass	Pass

Table 7 Stress Test 500 Nodes. These were our results after two distinct rounds when monitoring 200 nodes in a real time environment.

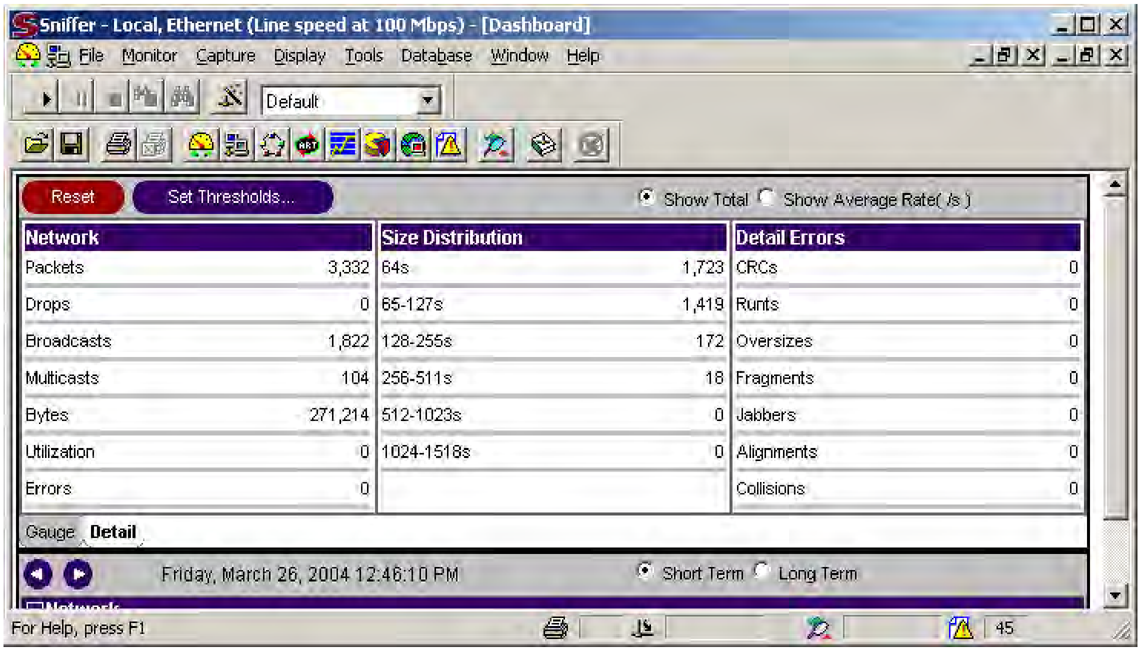
(500 Nodes)	**Round 1**			**Round 2**		
	24 Hrs	**48Hrs**	**76Hrs**	**24Hrs**	**48Hrs**	**76Hrs**
Microsoft 2000 Pro.	Fail	Pass	Pass	Pass	Pass	Pass
Microsoft XP	Fail	Pass	Pass	Pass	Pass	Pass
Solaris 8.0	Fail	Pass	Pass	Pass	Pass	Pass
Red Hat Linux 9.0	Fail	Fail	Pass	Pass	Pass	Pass

Table 8 Stress Test 800 Nodes. These were our results after two distinct rounds when monitoring 200 nodes in a real time environment.

(800 Nodes)	Round 1			Round 2		
	24 Hrs	48Hrs	76Hrs	24Hrs	48Hrs	76Hrs
Microsoft 2000 Pro.	Fail	Pass	Pass	Pass	Pass	Pass
Microsoft XP	Fail	Pass	Pass	Pass	Pass	Pass
Solaris 8.0	Fail	Pass	Pass	Pass	Pass	Pass
Red Hat Linux 9.0	Fail	Fail	Pass	Pass	Pass	Pass

Based on the above tables, we are confident that our program is robust. In round 1 of our testing, we ran into problems discovering bugs (mainly relating to threads and file input/output). Once we solved those, we were safely able to monitor the nodes without seeing any problems in performance for longer periods of time. Although unofficial, the program has been running for periods of days in our clients' lab with no signs of it weakening.

Since we also wanted to prove that our program would not cause an enormous amount of traffic, we captured real time network traffic and analyzed the information to present with this test phase. A lot of probing is done in order to present to the administrator the most up-to-date information on the status of nodes. This means a lot of packets being sent back and forth; our strategy was to send small packets in order to keep the network overhead reduced, while maintaining a high throughput. Looking at Figure 15, you can clearly see the size distribution of the packets is mainly 64 bytes.


Sniffer - Local, Ethernet (Line speed at 100 Mbps) - [Dashboard]
File Monitor Capture Display Tools Database Window Help
Default
Reset
Set Thresholds...
Gauge
Detail
For Help, press F1
45

● Show Total ○ Show Average Rate(/s)

Network		Size Distribution		Detail Errors	
Packets	3,332	64s	1,723	CRCs	0
Drops	0	65-127s	1,419	Runts	0
Broadcasts	1,822	128-255s	172	Oversizes	0
Multicasts	104	256-511s	18	Fragments	0
Bytes	271,214	512-1023s	0	Jabbers	0
Utilization	0	1024-1518s	0	Alignments	0
Errors	0			Collisions	0

Friday, March 26, 2004 12:46:10 PM ● Short Term ○ Long Term

Figure 15 Size Distribution of Packets. This illustrates how we focused on using small packets to reduce network traffic.

The ones that fall in the next category are the SNMP calls that we are making which requires values to be transferred. When calculating average packet size for this particular run of the program, you can see it ends up being approximately 81.39 Bytes long, which means very small packets are being sent rapidly with little overhead. When we looked at average rates per second, and had NAPS running with maximum number of threads, the sniffer told us we were dealing with around 26 packets per second.

Figure 16 also illustrates how careful we were in making sure that we were broadcasting small ICMP datagram packets. The green represents packets that are 64 bytes or less, and the blue is mainly for the SNMP, which again are slightly larger.

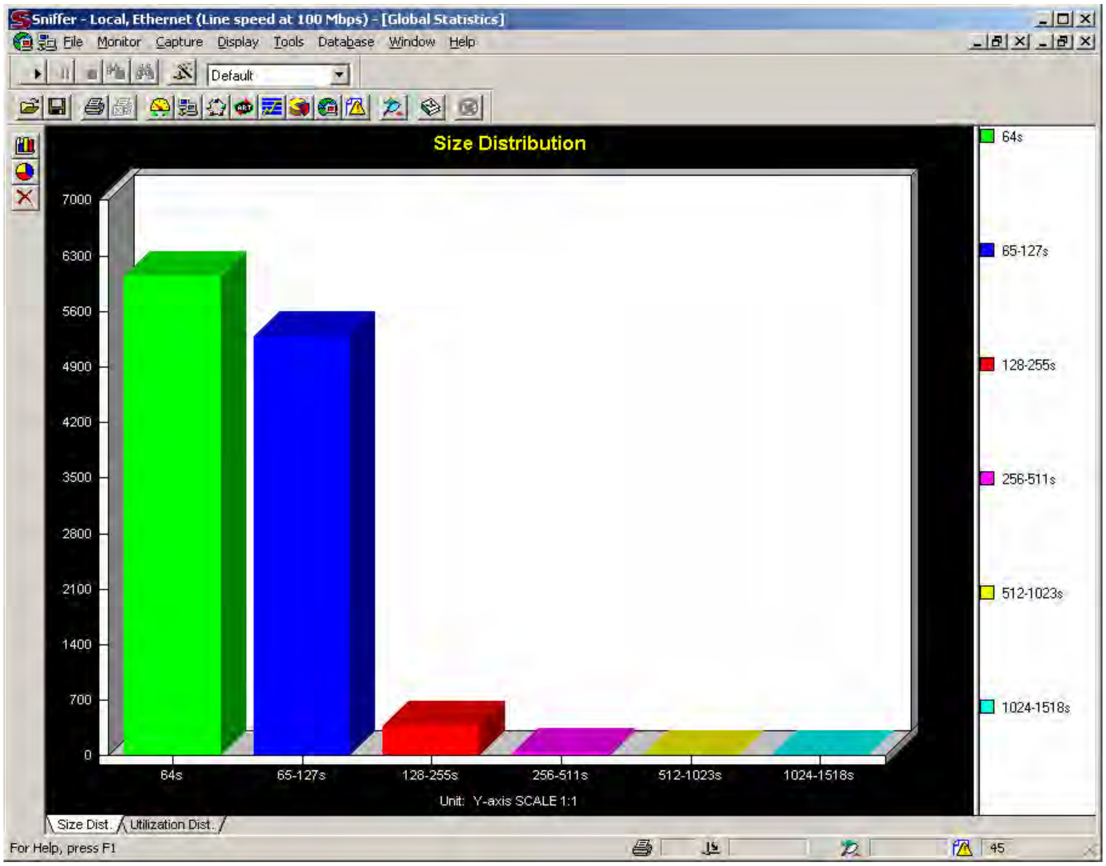


Figure 16 Packet Size Distribution. The 3D bar graph shows the distribution of packet sizes from our program when running.

Protocol distribution in seeing how efficiently our program was running was important to us. We wanted to visually see that we were only creating SNMP, ICMP, and DNS related queries. To verify this, we used Sniffer Pro’s Protocol Distribution tool to actively monitor an instance of one of our programs.

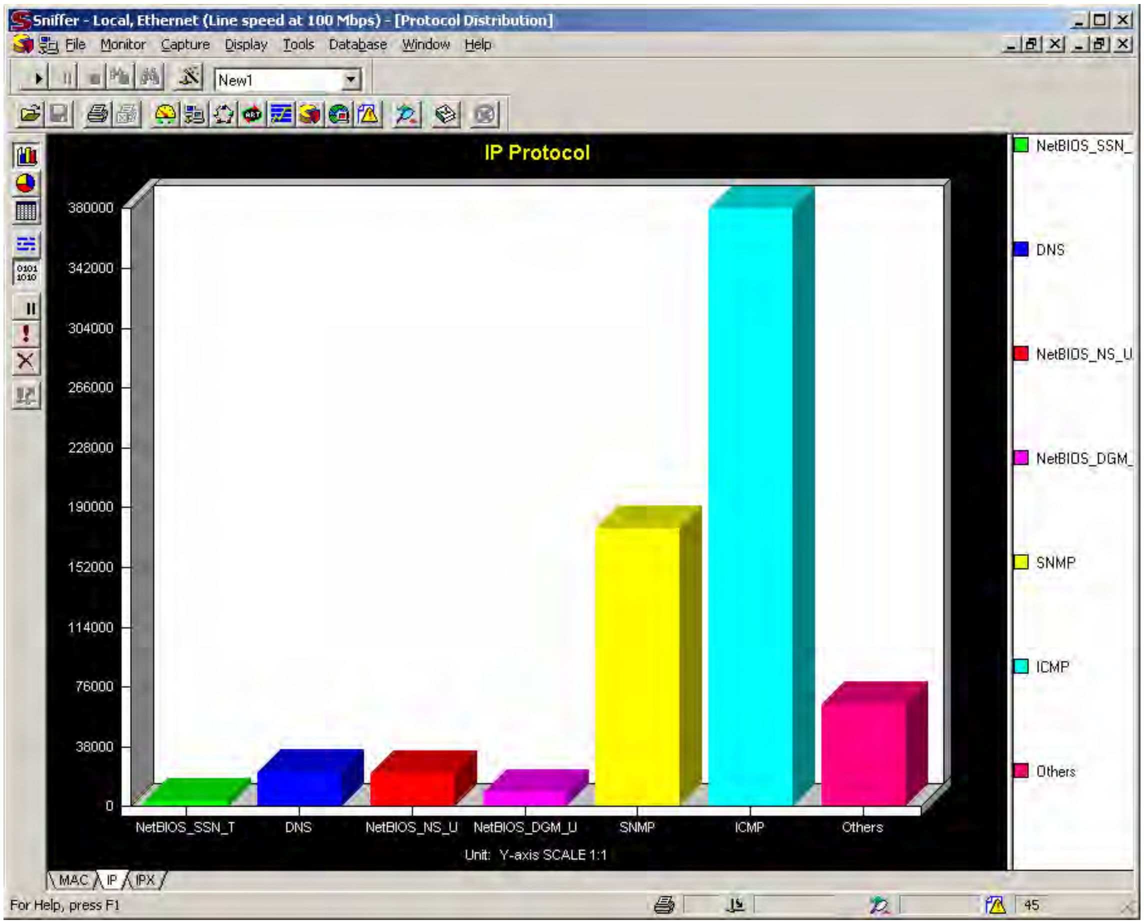
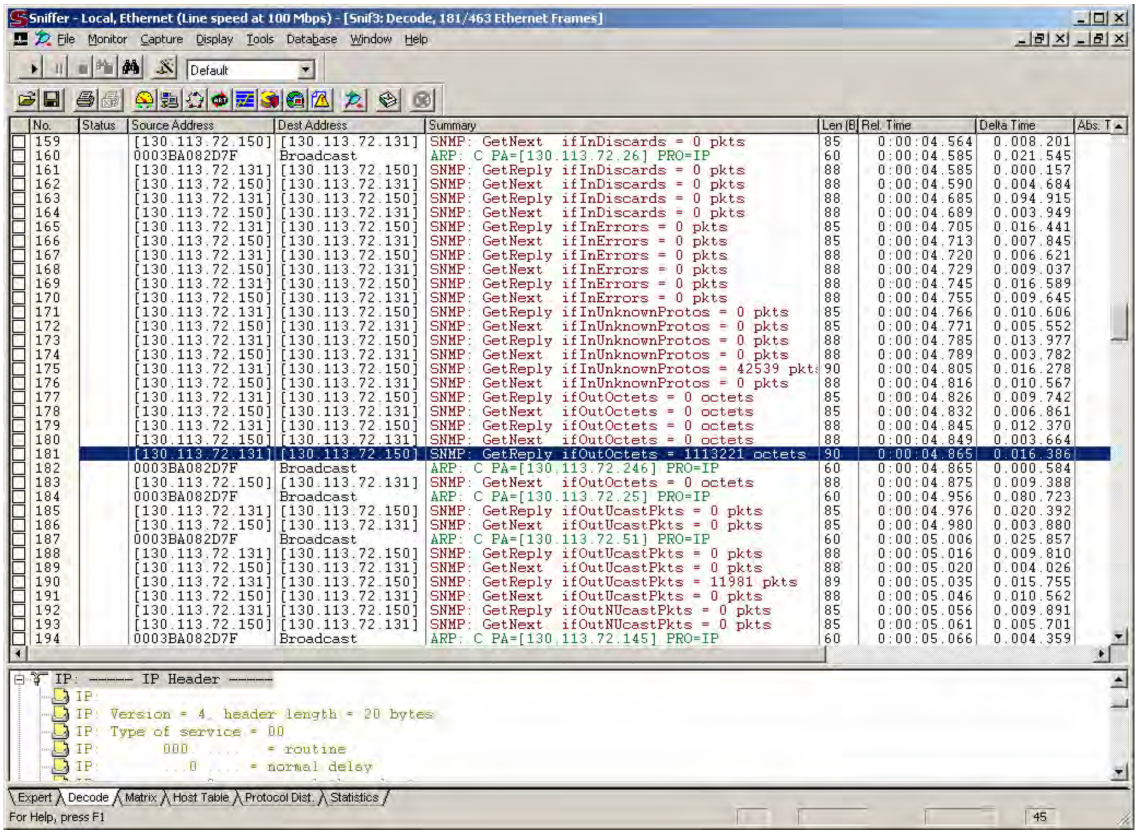


Figure 17 Overall Protocol Distribution. NAPS focuses on using strictly SNMP, ICMP and minimal amounts of either DNS, NetBIOS.

Figure 17 shows that our program is creating mainly SNMP and ICMP calls. Notice the little amount of NetBIOS and DNS querying that is done. In order to efficiently use the network, we do not continuously probe for new hostnames. Therefore, it is done once when the node is added and then only done at specific intervals, which are not short. This is due to the fact that most nodes do not have varying hostnames. Java also has its own cache that it creates, and does not rely on the system’s cache. This worked out to our advantage since once a hostname is entered into Java’s cache it remains there until the programs termination.

The other major component that can affect a network is the MIB Browser. This browser, especially with the “Get All” function can generate traffic. In order to verify that we were not generating an abundance of traffic, we verified its output usage with the sniffer. We ran a “Get All” on the node 130.113.72.132, paused the monitoring so that we were not broadcasting at the time, hence isolating our SNMP calls and verified the results. Below, Figure 18 verifies that the external library is functioning correctly and opening a connection to port 161 of node 130.113.72.132 and obtaining the information of all OID’s.

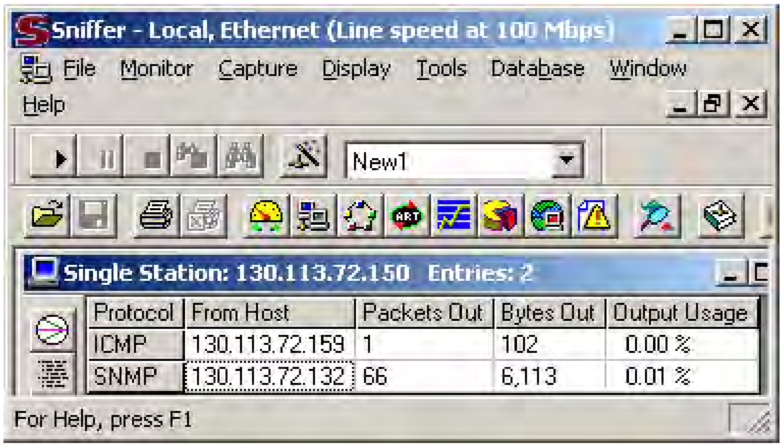

Sniffer - Local, Ethernet (Line speed at 100 Mbps) - [Snif3: Decode, 181/463 Ethernet Frames]
File Monitor Capture Display Tools Database Window Help
Default
Expert Decode Matrix Host Table Protocol Dist. Statistics
For Help, press F1
45

```
No.  Status Source Address   Dest Address      Summary                                       Len (B Rel. Time     Delta Time  Abs. T
159         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifInDiscards = 0 pkts          85     0:00:04.564   0.008.201
160         0003BA082D7F     Broadcast         ARP: C PA=[130.113.72.26] PRO=IP              60     0:00:04.585   0.021.545
161         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifInDiscards = 0 pkts          88     0:00:04.585   0.000.157
162         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifInDiscards = 0 pkts          88     0:00:04.590   0.004.684
163         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifInDiscards = 0 pkts          88     0:00:04.685   0.094.915
164         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifInDiscards = 0 pkts          88     0:00:04.689   0.003.949
165         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifInErrors = 0 pkts            85     0:00:04.705   0.016.441
166         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifInErrors = 0 pkts            85     0:00:04.713   0.007.845
167         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifInErrors = 0 pkts            88     0:00:04.720   0.006.621
168         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifInErrors = 0 pkts            88     0:00:04.729   0.009.037
169         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifInErrors = 0 pkts            88     0:00:04.745   0.016.589
170         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifInErrors = 0 pkts            88     0:00:04.755   0.009.645
171         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifInUnknownProtos = 0 pkts     85     0:00:04.766   0.010.606
172         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifInUnknownProtos = 0 pkts     85     0:00:04.771   0.005.552
173         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifInUnknownProtos = 0 pkts     88     0:00:04.785   0.013.977
174         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifInUnknownProtos = 0 pkts     88     0:00:04.789   0.003.782
175         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifInUnknownProtos = 42539 pkt  90     0:00:04.805   0.016.278
176         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifInUnknownProtos = 0 pkts     88     0:00:04.816   0.010.567
177         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifOutOctets = 0 octets         85     0:00:04.826   0.009.742
178         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifOutOctets = 0 octets         85     0:00:04.832   0.006.861
179         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifOutOctets = 0 octets         88     0:00:04.845   0.012.370
180         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifOutOctets = 0 octets         88     0:00:04.849   0.003.664
181         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifOutOctets = 1113221 octets   90     0:00:04.865   0.016.386
182         0003BA082D7F     Broadcast         ARP: C PA=[130.113.72.246] PRO=IP             60     0:00:04.865   0.000.584
183         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifOutOctets = 0 octets         88     0:00:04.875   0.009.388
184         0003BA082D7F     Broadcast         ARP: C PA=[130.113.72.25] PRO=IP              60     0:00:04.956   0.080.723
185         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifOutUcastPkts = 0 pkts        85     0:00:04.976   0.020.392
186         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifOutUcastPkts = 0 pkts        85     0:00:04.980   0.003.880
187         0003BA082D7F     Broadcast         ARP: C PA=[130.113.72.51] PRO=IP              60     0:00:05.006   0.025.857
188         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifOutUcastPkts = 0 pkts        88     0:00:05.016   0.009.810
189         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifOutUcastPkts = 0 pkts        88     0:00:05.020   0.004.026
190         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifOutUcastPkts = 11981 pkts    89     0:00:05.035   0.015.755
191         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifOutUcastPkts = 0 pkts        88     0:00:05.046   0.010.562
192         [130.113.72.131] [130.113.72.150]  SNMP: GetReply ifOutNUcastPkts = 0 pkts       85     0:00:05.056   0.009.891
193         [130.113.72.150] [130.113.72.131]  SNMP: GetNext  ifOutNUcastPkts = 0 pkts       85     0:00:05.061   0.005.701
194         0003BA082D7F     Broadcast         ARP: C PA=[130.113.72.145] PRO=IP             60     0:00:05.066   0.004.359

IP: ------ IP Header ------
  IP:
  IP: Version = 4, header length = 20 bytes
  IP: Type of service = 00
  IP:       000. .... = routine
  IP:       ...0 .... = normal delay
```

Figure 18 Sniffing out SNMP traffic. The figure shows part of a capture after executing the ‘Get All’ command in our MIB Browser. All the traffic in red are SNMP related.

Figure 19 illustrates the number of bytes that have been sent out on that same “get all” request that was made on the above screenshot.


Sniffer - Local, Ethernet (Line speed at 100 Mbps)
File Monitor Capture Display Tools Database Window
Help
New1
Single Station: 130.113.72.150 Entries: 2
For Help, press F1

Protocol	From Host	Packets Out	Bytes Out	Output Usage
ICMP	130.113.72.159	1	102	0.00 %
SNMP	130.113.72.132	66	6,113	0.01 %

Figure 19 Measuring SNMP traffic. The table shows the number of bytes and packets when completing a 'Get All' command in the MIB browser.

One can easily see this test phase really allowed us to verify that the code was working correctly. We were able to document all the low level activity using our Sniffer, and test it over long periods of time. The tests were done in different areas including: Cogeco subnets, complete ITB mappings, and various other campus wide computers.

Overall Success and Achieved Targets

After talking with our client and his history on giving us this project, we found that we were not the first ones to attempt this task. Two other groups in the past attempted to provide this tool to our supervisor. Both groups had run into several problems such as: lack of Linux knowledge, poor graphical frontend development abilities, and, overall, little networking knowledge. Our success relied on our ability to use each others strong points. No one person in this team would have been able to succeed on their own in completing a project of this nature. Matthew's knowledge and passion for UI development resulted in a robust and user friendly environment. Paul's knowledge on thread controls and our SNMP external library allowed us to populate the frontend with correct data. Tim, who successfully dealt with bridging both Paul and Matt's work together, involved a lot of thought on issues such as concurrency and optimization. Finally, Nicholas's networking knowledge and system knowledge provided the group with many answers to problems that were occurring throughout the year, allowing the others to focus strictly on development. Together, all of our initial goals that we had set for ourselves were satisfied, but above all, we met our client's expectations.

For the Future

The ability to continue with this project is something that should be done. It is rare to find a free networking monitoring tool of this size on the internet. Most will be trial editions. The fact that it runs on all platforms and it maintains a proper structure means that other groups in the future should easily be able to work off of it and add components such as the ones marked as incomplete in our scope. We do not intend to stop working on the project as a whole, even after the term is complete. Since we all enjoy programming as a pastime, we are sure to see more revisions being released with newer code, including more functionality for any network administrator. The top priority functions that we would like to see complete are: automatic network discovery and the ability to load any MIB module file the user desires to increase the MIB's power.

Appendix A – System’s Setup

Online Forum, Email List, and official development Site

As mentioned in the Scope and Issue Tracking section all our group communication and issues were logged. This allows us to keep track of who said what and what programmer will be held responsible for completing certain task. The links to these two sites that we keep open for anyone to review are the following:

Email Listings:	http://server791.dnslive.net/pipermail/4zp6-naps_creativestudent.com/
Forum:	http://www.creativestudent.com/naps/
JavaDocs:	http://www.creativestudent.com/naps/

All the minor bug fixes that needed to be assigned were done there either via email or the forum if the issue was considered to be major by the programmer. Our JavaDocs and packages were also maintained periodically there for anyone to review or download.

Network File Server Setup

Many remote installations of operating systems needed to be completed in order to successfully demo our final product for the year end presentation. Below are brief steps explaining how the NFS server was created. Please note that the NFS server was created on a Red Hat Linux 6.2 operating system. If you want to review a sample setup please log into the host named ‘4zp6Server’ who’s IP address is: 130.113.72.160. On this computer the base Slackware image is located in /Slackware and is NFS ready.

- /etc/exports needed to be modified to specify what directory you want to allow to be mounted.
- Once that was done you must modify the /etc/host.deny and /etc/host.allow files to specify who is allowed to mount your local directory. You can choose to either allow a whole subnet, or specific nodes.
- Once that is complete you must start or re-start your service by issuing commands like:

  service nfs start
  service nfs restart and then
  service portmap restart
- Then on the client (or remote) machine you must also start the service by doing the following:

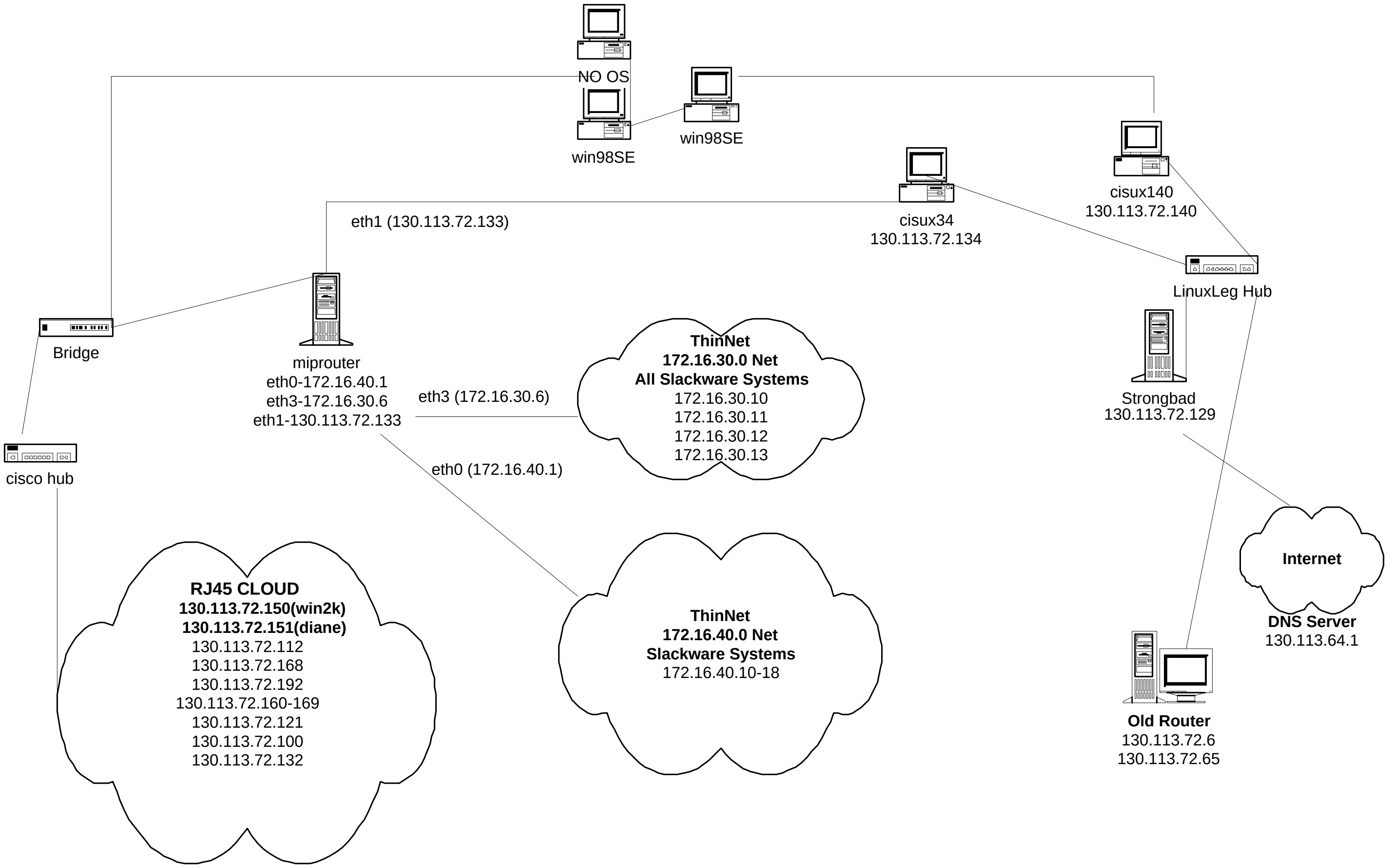
  service nfs start or
  service nfs restart
- Then to mount the directory from the server you would do something like the following:

```
mount –t nfs <systemName>:<remoteDir>  <mountPoint>
i.e. mount –t nfs 4zp6Server:/slackware  /Slackware
```

Slackware & Red Hat Linux Setup/Installation

Due to our group's strong preference for Red Hat Linux we decided with the majority of the machines to complete a clean install of Red hat 6.2. In order to be backward compatible with some of the older machines we also installed Slackware version 3.3. The reasons for this is that the older 386's and 486's located in the lab have old kernels (version 2.x) of Slackware with no compilers installed on them. If the need arises where we need to install a package of some sort on those machines then we would be able to statically compile the program on the newer versions and transfer them over. Many problems came around when re-installing many of them mostly due to hardware failures. We will not document any further on the installations procedures as they are already well explained elsewhere.

Subnets of Demonstration Setup



Explanation to Demonstration Setup

One of our initial goals for this project was to ensure that we were not limiting ourselves to any one particular environment. It was critical that we knew that our program could run on any platform and in a real life network situation. This product is not geared for a small Windows X environment, but rather for any mid to large scale network that holds any varying types of hosts. Therefore by looking at the diagram in the previous section it is clear that we setup the environment to monitor many different types of machines. The demo will be concurrently running on three OS:

UNIX Solaris 8.0
Microsoft Windows 2000 Professional
Red Hat Linux 9.0

These three demos will all have the same mapping loaded of the lab (above) which means that at least the following environments will be monitored:

UNIX Solaris 8.0
Microsoft Windows 2000 Professional
Microsoft Windows 98 SE
Microsoft Windows 98
Microsoft Windows 95
Red Hat Linux 9.0
Red Hat Linux 6.2
Slackware Version 3.3
Slackware Version 7.1
Mandrake Linux

Clearly we can conclude that we tested our program in a “real-world” environment successfully.

Windows SNMP Setup

To setup SNMP on older flavours of Microsoft Windows we needed to find and download an SNMP agent listener. We decided to use SNMP4tPC, which can be obtained from: http://www.wtcs.org/snmp4tpc/. With these files downloaded we proceeded to installing it using the Network Properties window, and adding a new service.

Net-SNMP Setup

Red Hat 6.2

- Used the binary RPM package version: net-snmp-5.0.9-4.src.rpm
- To install package I typed the following:

```
rpm –i net-snmp-5.0.9-4.src.rpm
```

- Packaged is installed in /usr/local/ by default

- To activate snmpd you can goto /usr/local/sbin or for future convenience add /usr/local/sbin to your .bash_profile PATH.

Slackware

- Net-SNMP is available from: http://net-snmp.sourceforge.net
- Download ucd-snmp-4.2.1.tar.gz (or a later version when available) to your LINUX system root directory.
- Unpack the zipped tar file using:

```
tar  xzfv ucd-snmp-4.2.1.tar.gz
cd /ucd-snmp-4.2.1
```

- Configure the package to include the host MIB support.

```
./configure --with-mib-modules=host
```

- Generate the package: make
- Install the runtime components:

```
make install
```

- Manually start or stop the snmp daemon:

```
/etc/rc.d/init.d/snmpd start
/etc/rc.d/init.d/snmpd stop
```

The above installs the environment but then several configuration files need to be modified in order for the daemon to actually successfully work. These files took a while to create and modify correctly (Specifically the /etc/snmpd.conf file).

WinCVS Setup Steps

In order to be more productive and maintain issue tracking we needed to use CVS. The repositories were created for us on the school servers. From there it was up to us to maintain its structure throughout the year. Since the majority of our coding was done at home we wanted a quick and efficient way to check modules in and out over Secure Shell on our school accounts. The solution involved using a program called WinCVS. This tool provided a graphical front-end which made our development much easier and faster. Our repository that was created was located at /u1/cs4zp6/cs4zp6gb/. In that repository is the CVSROOT directory that contains what are often referred to as the “administrative files.” The directory in there is called NAPS which is the module we will be working with. Never should any of us try to navigate and modify/delete these files/folders located in these directories/subdirectories. The reason is that you will destroy the history files, and the rights of the modified files will be changed to your ownership. The following modules were created to be easily checkout and to maintain independent structures within our repository:

Doc	CVSROOT/NAPS/doc
Epp	CVSROOT/NAPS/epp
poConcept	CVSROOT/NAPS/poConcept
source	CVSROOT/NAPS/source
writeUP	CVSROOT/NAPS/writeUP

Below are the steps to get the program running correctly under the McMaster Environment:

- Download winCVS 1.3 from http://www.wincvs.org/
- Download SSH for windows from www.ssh.com
- Log in to your account with SSH like you guys always have. Then type
- "pico .cshrc"
- DO NOT TOUCH ANY LINES THAT MIGHT BE IN THERE…at the bottom or on some separate line type in the following line
  ”setenv CVSROOT /u1/cs4zp6/cs4zp6gb”
  then type in crt-O and then crt-X to exit
- Open up winCVS 1.3 (should be an icon on the desktop)
- Under "Admin-Preference"
- GENERAL TAB:
- Authentication: ssh
- Click on settings and check off SSH client and type in ssh2 in the blank space underneath. Click 'ok'
- Path: /u1/cs4zp6/cs4zp6gb/
- Host Address: ritchie.mcmaster.ca
- User name: <loginID
- Which means CVSROOT should look something like
  merizzn@ritchie.mcmaster.ca:/u1/cs4zp6/cs4zp6gb/
- GLOBALS Tab -- Leave alone.
- CVS Tab -- Create a directory locally that you want your files downloaded to
- WINCVS Tab -- Default viewer just type in 'notepad'
- COMMON DIALOGS Tab -- Leave alone.
- You are now ready to proceed to completing a Checkout:
- "Remote-Checkout Module" menu…
- CHECKOUT SETTINGS:
- Under module name and path on the server just click on the "…" next to the drop box… What this will do is try to connect to our repository and check to

see what Modules are available for you to checkout. You click on "…" (might have to click "refresh afterwards) and it will attempt to connect.. It will prompt you for your password and then return a list similar to the one above in my email here.

- From there just simply click on the module you want to check out! The first one CVSROOT will checkout the whole project… try to avoid doing this… just narrow it down to the module you want… in this case it should be poConcept.
- just click on 'Ok' and it will download all of the directories content locally into the directory you specified in step 5j.
- Now you are all set… just work locally. When you are ready to commit your files (i.e done working) just right click on it and click on "commit"…
- If it's a new file.. you must first "add it" (Modify menu- then "add It") then commit it like above.
- **PLEASE AT FIRST SAVE LOCALLY AND MAKE A COPY OUTSIDE THE HOME DIR YOU ARE WORKING ON**
- Once you committed the file use SSH on its own to browse to the directory to see if you really did commit your new file/updated file.

Committing/Updating/Working Concurrently

When working concurrently with the other 3 members in the group you will run into the problem that everyone will check out a file at the same time. To prevent people from saving work on top of others you MUST get use to checking for updates before committing. If you don't update your local copy before committing it you will overwrite your partner's work.

WinCVS steps

- Checkout your project like you would normally
- Work locally
- When completed the files under winCVS will appear Red and it will say "Mod. File" under status. Before right clicking on the file and "committing it" you must select "Update" first. This will go online and check for a new revision. If the revision number has changed it will compare the difference between the two versions and update your file locally on your computer along with any modifications you might have put locally. So this means that no one's work is lost.
- Once you have "Updated" the file you can then proceed to "Committing" You must do these steps one after another quickly or else someone can overwrite the update two seconds before you commit (CVS's major drawback-No locking).

CLI Steps

- Cvs checkout poConcept/blackboard.java
- Cs poConcept in home dir
- Work locally on file
- Cvs update
- Cvs commit –m "I updated the file now I'm committing the file with my changes"
- Cd ..
- Cvs release –d poConcept

List of important Commands Used/Needed

The clients' lab was given to us with minimal instructions and only one or two restrictions. This left us with many decisions for network design. Nicholas Merizzi was in charge of making those decisions and implementing them. Unfortunately in the world of computing there are yet to be many standards between operating systems to be established. A 'route' command in UNIX will differ from a 'route' command in Linux. Everything from routing tables, to network interface card setups vary from system to system. Below is a compiled list of some of the key commands that were needed to complete the setup:

Linux:

- To add the default Gateway: *route add default gw <gwIp>*
- To add the localnet to the routing table you had to modify the /etc/networks file. The file should at least have two lines (one for the loopback and one for localnet)

  route add localnet

Solaris8:

- To add the default gateway:

  route add default 130.113.72.129 1

  or *route add default 130.113.72.129 eri0*
- To get the DNS servers setup you had to modify the /etc/resolv.conf so Diane's resolv.conf looks something like this:

  ```
  domain mcmaster.ca
  nameserver 130.113.64.1
  nameserver 130.113.68.10
  ```
- To delete a network from the routing table on unix:

  route delete –net <netIP> <gwIP>

Linux:

- I needed to change the mouse speed on a couple of the linux boxes that we were using because it was going way too fast! Here is the command for it:

  xset m 1 30
- */etc/init.d/snmpd start* to start the demon
- *snmpwalk 127.0.0.1* to test if snmpd has started… should return a ton of OIDs with their proper value

Mandrake Linux:

- To execute binaries in Mandrake you need to type:

  bash <filename.bin>

Solaris UNIX:

- To add the localnet in UNIX:

  route add –net 130.113.72.0 diane 0

Debian Linux:

- Then to add the default gateway (Strongbad) the following command is needed:

  route –n add default 130.113.72.129

Creating an Executable JAR file

- You cannot create a JAR inside of JAR file so please extract all other JAR's into their corresponding directories.
- Inside /sourcenew directory create the manifest file. Call it manifest to be clear.
- Then open a terminal window and navigate to your ../sourcenew directory and type in:

  ```
  jar cmf manifest NAPS.jar *
  ```
- Then to execute it on any platform type in the following

  ```
  java –jar NAPS.JAR
  ```
- To extract a jar File you would type the following

  ```
  jar xf <filename.jar>
  ```
- In Windows we also used a JAR to EXE converter program to create a win32.exe program. This program can be found at:

  http://mpowers.net/product.html

Appendix B - Users’ Guide to NAPS

NAPS

Not Another Port Scanner

User's Manual

About:

NAPS was created by Nick Merizzi, Matt Picheca, Paul Paszynski, and Tim Tisdall as their final year project in the computer science program at McMaster University in 2003-2004. W.F.S. Poehlman was the faculty advisor for the project.

Contents

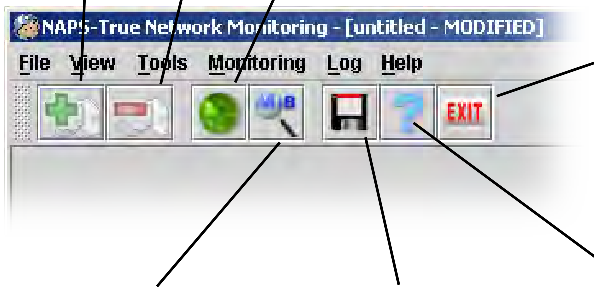
Quick Reference Guide

Main Toolbar

Add Node: This button will bring up a small dialog that will allow you to add nodes to the window by their IP address.

Delete Node: After clicking this button, you'll be shown a small window where you can delete nodes currently displayed in the main window.

Polling On/Off: When the button displays ● then NAPS is actively getting information from every node added to the window. If you click on this button it changes to ● and then NAPS will no longer be getting new information. If you click on it again, then it will return to active polling.



Exit: Exits the program, but will prompt you to save your mapping if you have not already done so.

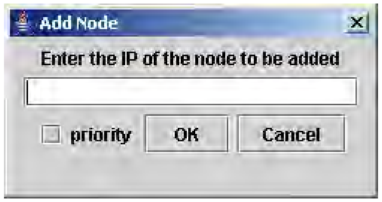
Help: Brings up the help window.

MIB Browser: Opens the MIB browser window so you can make SNMP queries on different computers.

Save: Save your current mapping. (disabled if there are no nodes in the winodw)

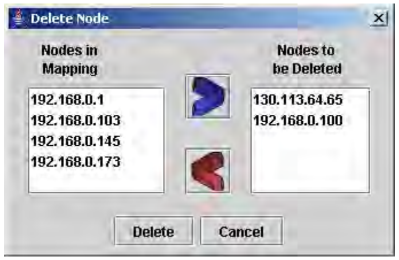
Adding Nodes

Clicking on the Add Node button ➕, will bring up the window shown on the right. You can enter the IP of a machine you want to monitor or you can enter a range of IPs. (ex. 1.2.3.4-6 will add the nodes 1.2.3.4, 1.2.3.5, and 1.2.3.6) If you check off the "priority" box, by clicking on it, then you will be notified by email when something important happens to the machines you entered.



Deleting Nodes

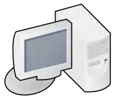
Clicking on the Delete Node button ➖, will bring up the window shown on the right. You can select the IPs of the machines you wish to stop monitoring on the left and then click the ➡ arrow to move them to the right. When you click "Delete" then all of the nodes listed on the right will be removed.



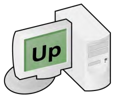
Nodes in Mapping: 192.168.0.1, 192.168.0.103, 192.168.0.145, 192.168.0.173

Nodes to be Deleted: 130.113.64.65, 192.168.0.100

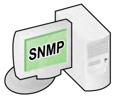
Interpretting the Screen



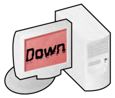
Node has been added to the screen but no information has been retrieved yet.



The node is responding to pings, but doesn't have an SNMP agent.



The node is responding to both pings and to SNMP requests.



This node is not responding to ping.

Requirements and Installation

Requirements

NAPS was deisgned entirely in Java and can operate on any machine running Sun OS, Linux, Windows (2000, NT, or XP), or most flavours of Unix. Also, in order for the program to work properly the user running the program must have sufficient access to be able to run the "ping" command. You can test if you meet this requirement by going to the command line and typing "ping 127.0.0.1". If the message returned indicates that you don't have access to ping then NAPS will not work properly for you. Talk to your system administrator to gain access to the ping command if you do not already have it.

Installation

If you do not already have the **Java Virtual Machine**(JVM) then you may install Java 1.4.2 which is found on the CD (it should be clearly indicated as to which file that is). Alternatively, you may download the latest version of Java from http://java.sun.com with your web browser.

As mentioned in the requirements, you must have **access to the ping command**. If you don't have access to ping then it's most likely due to a security policy in effect on your computer that can only be changed by the system administrator.

The installation of NAPS is fairly simple because the program is packaged into a JAR file. Simply copy the NAPS directory off of the CD to where you'd like to run the program. Make sure that after copying the files you change the permissions on all the files in the directory and the directory itself so that it is not read-only (ie has write access on). The reason for this is that NAPS will create log files and save network mappings to that directory.

To run the program in Windows, most systems will be set up after installing the JVM to allow you to double-click on the "naps.jar" file to execute the program. Alternatively, you can run the program from the command line by first making the NAPS directory the current directory and typing "java -jar naps.jar". If you're running Sun OS, Linux, or some other Unix similar system, you can run the program by typing "java -jar naps.jar" at the command line when in the NAPS directory.

Program User's Guide

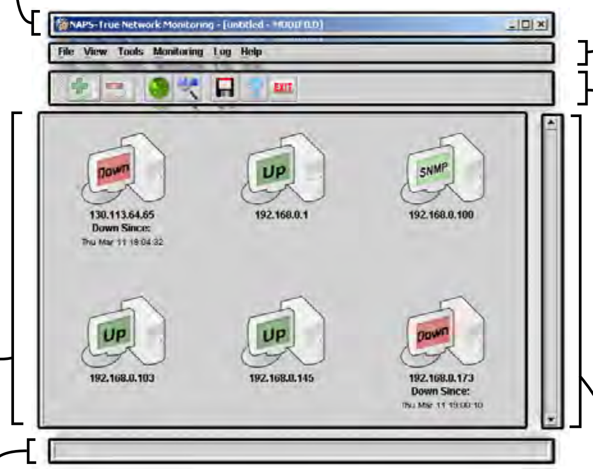
Main Window

Thank you for using NAPS. We hope that you find the program easy to use and useful for monitoring your own computer network. The figure below will give you a brief general description of the different aspects of the main NAPS window. The specifics will be covered in subsequent sections with the exception of the toolbar which is covered in the "Quick Reference" section. Also, the meaning of the machine icons should be mostly self-explanatory, but are also explained in the "Quick Reference".

Title Bar: This changes depending on the operating system you're using

Menu Bar: Gives you access to almost all of NAPS' functions

Main View: This is where all of the nodes are displayed and where new nodes are added to.



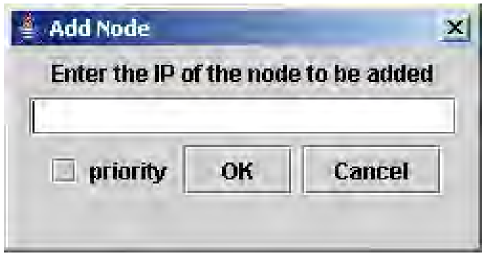
Tool Bar: Quick way to access the main aspects of NAPS

Status Bar: Gives you information in some circumstances about the program's activities.

Scrollbar: When there are a lot of nodes in the window this will become active.

Add Node Window

To monitor a machine over the network, you first need to add it to the main window with the Add Node window. To bring up this window, you can either click the Add Node button (➕), press CTRL-A on the keyboard, or select *Add Node* from the *Monitoring* menu item.



After clicking the Add Node button in the main toolbar you will see the Add Node Window shown here.

Now that you have the window open, you can begin to add nodes by entering their IP addresses. When you've entered the IP address that you wish to be added, click the ***OK*** button. Additionally, if you'd like to be notified about events that occur involving this IP by email and also have the information logged in a special log file then you can check off the *priority* box before clicking ***OK***.

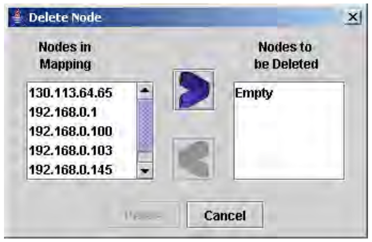
As well as simply entering a **single IP address** in the Add Node window, you can also enter a **range of IP addresses**. To enter a range, you specify the first three bytes of the IP address explicitly and then you may enter a hyphenated number range as the last byte. For example, lets say you'd like to add all of the computers in your local area network and you know that the computers range from 192.168.0.100 to 192.168.0.150. To enter all the computers between those two ranges you would enter 192.168.0.100-150 in the Add Node window. Please note that the last byte of the IP address is the only one that you're allowed to specify a range on.

Delete Node Window

If you click on the ➖ icon, press CTRL-D, or select *Delete Node* under the *Monitoring* menu item, you'll see a window similar to the one in top figure.

When you first bring up the Delete Node window, you'll see a list of all the nodes currently added on the left and another similar field to the right that says "Empty". You delete nodes by selecting on the nodes you wish to delete on the left side and then clicking the ▶ arrow to move them into the field titled "Nodes to be Deleted". If you accidentally add a node to the right box that you don't want to delete, simply click on it and click on the ◀ button to move it back to the left box. When you've finished moving nodes into their appropriate boxes (ie the ones you want to keep are in the left box and the ones you want to delete are in the right box), click on ***Delete*** to commit your changes.

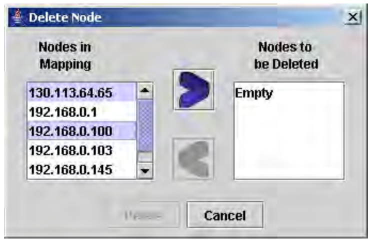
When selecting items in the boxes, you can do it in multiple ways. If you'd like to select multiple items in the list, you can hold down the CTRL key and click on each of the items you wish to select. Also, if you wish to select a range of nodes, you click on the first node in the range, hold down Shift, and then click on the last item in the node. When you use the Shift method of selection then all nodes between the two you select will be selected as well.



130.113.64.65
192.168.0.1
192.168.0.100
192.168.0.103
192.168.0.145

Empty

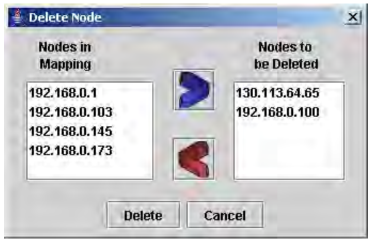
When you first bring up the Delete Node info you should see a screen like this.



130.113.64.65
192.168.0.1
192.168.0.100
192.168.0.103
192.168.0.145

Empty

You can select multiple IPs by holding down the CTRL key and clicking on them.



192.168.0.1
192.168.0.103
192.168.0.145
192.168.0.173

130.113.64.65
192.168.0.100

After selecting the nodes you want to delete, clicking the right arrow button moves those addresses to the right box. In this case, the two IPs on the left will be deleted when the ***Delete*** button is clicked.

Menu Items

Each menu item can be accessed by clicking on them. You can also use the keyboard by holding down `Alt` and typing the letter underlined in the name of the item you wish to select. There are also quick keys defined for some menu items which are listed just to the right of those items (also see *Keyboard Shortcut Reference*).

New Mapping clears all the nodes in the current window so you can start with a blank screen

Open Mapping brings up an open file dialog window so you can select a saved mapping and load it into the main window.

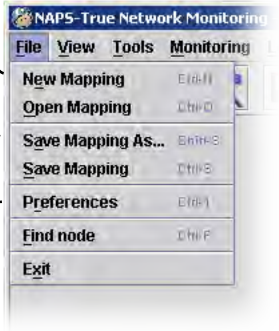
Save Mapping As... brings up a save file dialog and allows you to type in a file name to save this mapping

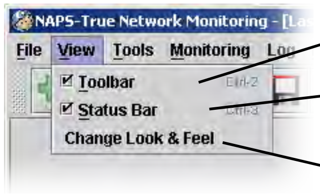
Save Mapping will save to the current file, or if there is none then will bring up a save file dialog so you may type in a file name. (the current file name is displayed in the title bar in between the square brackets)

Preferences brings up the preferences window (See "Setting Preferences")

Exit Exits NAPS

Find node brings up a dialog that will allow you to search for an IP in the current mapping and then highlight the node if found.

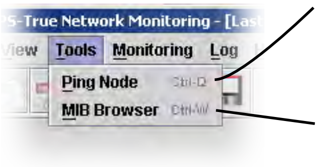




Toolbar turns the icon toolbar on/off

Status Bar turns the status bar at the bottom of the window on/off

Change Look & Feel allows you to alter the current look of NAPS by selecting an alternate application "skin"

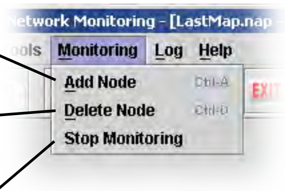


Ping Node brings up a dialog to allow you to ping any IP or hostname address. The address need not be one currently in the mapping. (see Ping an Address)

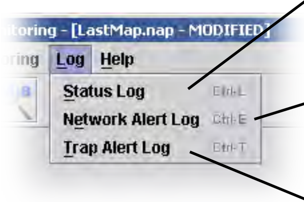
MIB Browser brings up the MIB browser window (see MIB Browser)

Add Node brings up the Add Node dialog to allow you to add a new node to the mapping (see Add Node Window)

Delete Node brings up the Delete Node dialog which allows you to delete any nodes in the current mapping (see Delete Node Window)



Stop Monitoring this tells NAPS that you want to stop all monitoring of the nodes. In other words, NAPS will stop getting SNMP information and stop pinging nodes and no new information will appear in NAPS.

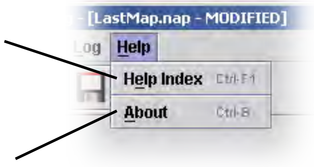


Status Log brings up a window showing the contents of the status log. This window is updated as new information is found.

Network Alert Log similar to the Status Log, but only gives information on nodes marked as "priority"

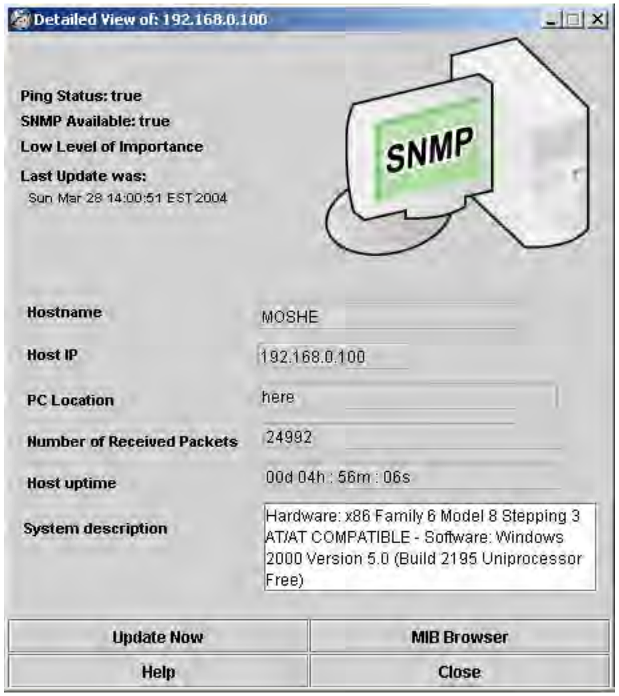
Trap Alert Log lists all the SNMP traps captured from nodes added to NAPS

Help Index brings up the help window which contains information on how to use NAPS



About gives information on the creators of NAPS. To close this window, simply click on it.

Node Information Window



Detailed View of: 192.168.0.100

Ping Status: true

SNMP Available: true

Low Level of Importance

Last Update was: Sun Mar 28 14:00:51 EST 2004

Hostname: MOSHE

Host IP: 192.168.0.100

PC Location: here

Number of Received Packets: 24992

Host uptime: 00d 04h : 56m : 06s

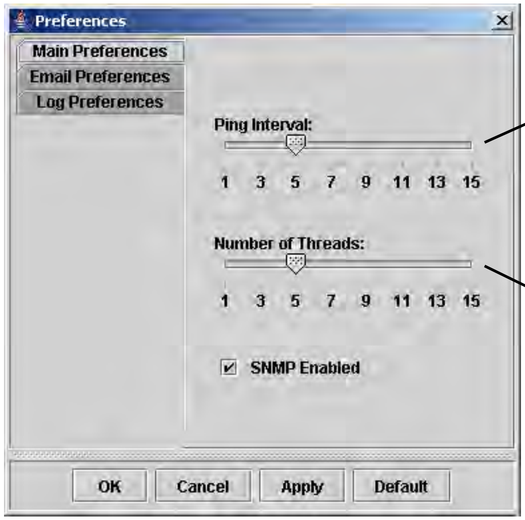
System description: Hardware: x86 Family 6 Model 8 Stepping 3 AT/AT COMPATIBLE - Software: Windows 2000 Version 5.0 (Build 2195 Uniprocessor Free)

The Node Information Window is shown in the figure above. Most of the information and buttons should be self-explanatory. However, the uer should be aware of the following facts about NAPS:

A node may appear to go up and down if the response time to pinging varies between less than 1 second and greater than 1 second. This is due to the fact that the ping gives up waiting for a response after 1 second.

If a machine is not responding to pings then it is automatically assumed that the machine will also not respond to SNMP requests. While this is not always the case, it is true in most cases. If you know specifically that a machine that is shown as "Down" is actually connected and will respond to SNMP, then you can get the SNMP information through the MIB browser (if you click on the ***MIB Browser*** button then you will see the MIB Browser with this machine's address already entered).

Setting Preferences



Preferences

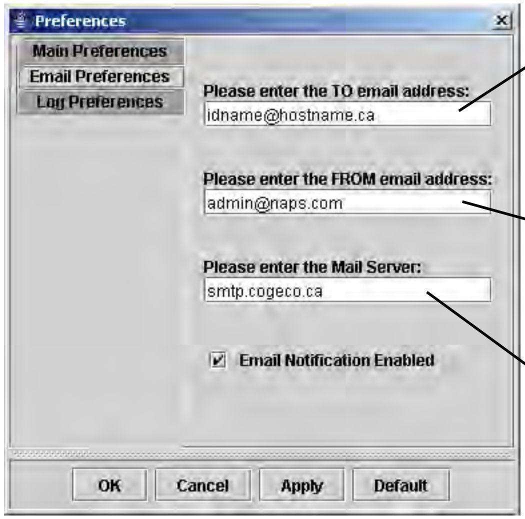
Ping Interval: 5

Number of Threads: 5

☑ SNMP Enabled

Set the ping interval in seconds. (the time to wait between successive pollings)

Set the number of Threads the backend should use to update the nodes in the main window.



Preferences

Please enter the TO email address: idname@hostname.ca

Please enter the FROM email address: admin@naps.com

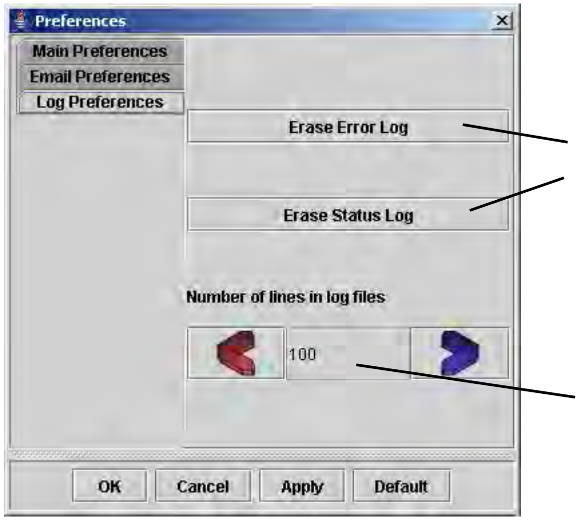
Please enter the Mail Server: smtp.cogeco.ca

☑ Email Notification Enabled

Who should be notified about important events? Put their email address in here.

Who should the emails sent appear to be from?

Place an SMTP server address that's accessable from this machine.

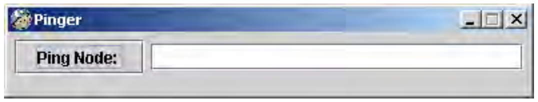


Number of lines in log files: 100

These two buttons will erase all the contents of their respective files.

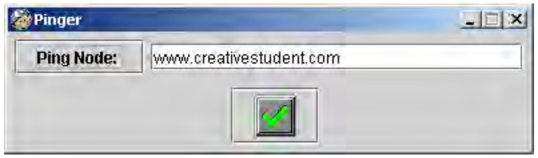
How many lines should be kept in the log file?

Ping an Address



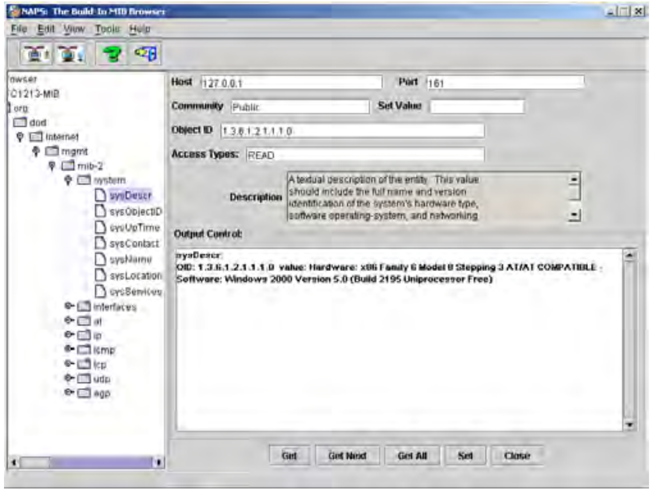
When selecting *Ping a Node* from the *Tools* menu you will be shown a dialog box where you can enter a machine address. Unlike the Add Node window, you can enter either the IP address or the machine's Hostname. When you click on the ***Ping Node:*** button on the left of the window, the program will try to ping that machine to see if it's responding. The result of the ping will be shown just below the textfield where you entered the machine's name (IP or hostname). If you see a ✓ then that machine has responded to the ping and if you see a ✗ then that means the machine didn't respond to the ping.

Please be aware that a machine that doesn't respond to ping is not necessarily disconnected from the network. There is a possibility that the response from the ping is taking longer than 1 second which will show falsely as a ✗ because the ping gives up waiting for the response after 1 second. Also, due to network security concerns, some system administrators have set up their machines to not respond to ping to avoid "denial of service" attacks.



www.creativestudent.com

After entering an address and clicking Ping Node: we get a positive response



Host: 127.0.0.1 Port: 161

Community: Public Set Value:

Object ID: 1.3.6.1.2.1.1.1.0

Access Types: READ

Description: A textual description of the entity. This value should include the full name and version identification of the system's hardware type, software operating-system, and networking

Output Control:

```
sysDescr:
OID: 1.3.6.1.2.1.1.1.0  value: Hardware: x86 Family 6 Model 8 Stepping 3 AT/AT COMPATIBLE -
Software: Windows 2000 Version 5.0 (Build 2195 Uniprocessor Free)
```

The MIB Browser window is probably the most complicated to use window in the NAPS program. For most casual users of NAPS you will not need to use this window, but more advanced users will find it useful to get additional information the Node Info window doesn't have. Also, the MIB Browser will also allow you to set values if you have the authorization (ie. you have set the "Community" field to a community that has permission to write values on this particular machine).

> **!** **NOTE:** You will do no harm to your network or the machines on your network if you just use the "Get" buttons. However, setting values may cause problems on either particular machines or on the network in general. Essentially, setting values should be left to users who know what they're doing.

Keyboard Shortcut Reference

Ctrl - N	New Mapping
Ctrl - O	Open Mapping
Shift - S	Save Mapping As...
Ctrl - S	Save Mapping
Ctrl - 1	Edit User Preferences
Ctrl - 2	Toolbar On/Off
Ctrl - 3	Status Bar On/Off
Ctrl - Q	Ping an address
Ctrl - W	MIB browser
Ctrl - A	Add Node(s)
Ctrl - D	Delete Node(s)
Ctrl - T	Delete Node(s)
Ctrl - L	View Status Log
Ctrl - E	View Network Alert Log
F1	User Help Menu

Additionally, all of the standard shortcuts specified by your operating system for cuting, copying, and pasting text should work in text fields. Other shortcuts may work too, but are operating system dependent.

Appendix C - Phase 1 Questionnaire

In order to get the proper feedback we selected eight people to test our program. To respect those individuals confidentially there names will not appear in our results. Below is the process we followed to gain outside perspective on our CHI design and possible bugs.

What is the user's computer level (Beginner/Intermediate/Advance)?

Provide the user with either the CDROM or the URL with the Software package.

Was user able to Install and execute program? If any feedback given please specify below:

What are the user's immediate first impressions once opening the software?

Was the user able to:
(a) Find the help menu:
(b) Pleased with the Help menu:

Have the user add nodes, have him/her add a single node as well as a range. Have him/her add nodes two different ways. Any problems completing the task?

Have them find two ways to delete a specific node in that range. Any problems completing the task?

Have the user save the mapping, clear the mapping, and then reload the mapping that he/she just saved. Any problems completing the task?

Find two ways to open the MIB Browser, can the user explain the difference with the two ways?

Have user get some random OID's, does the user have any complaints regarding the output?

To check retention over time ask the user 48 hours later if he remembers how to add a range of nodes, and what the MIB browser allows him/her to do.

Overall feedback (Please let user write-in on his/her own):

Appendix D - Phase 2 Test Module Used

```
/**
 * blackboardTest.java
 *
 * Created on March 13, 2004, 2:56 PM
 *
 * @author  Paul Paszynski
 */
import java.util.*;
import junit.framework.*;
import blackboard.*;
import java.net.*;
import java.lang.*;


public class BlackBoardTest extends TestCase {

    //Make sure this is an even number and is >=2 or it will fail!
    public static int MaxNodes = 1000; //defualt max nodes

    public static BlackBoard BBMethods;
    private String ipArray[] = new String[MaxNodes];
    private int numip = 0;
    private InetAddress tempInet;
    private LanNode tempNode;
    private LanNode nodeArray[] = new LanNode[MaxNodes];
    private int upArray[] = new int[MaxNodes];
    private static int numNodesToAdd;

    public BlackBoardTest(String name) {
        super(name);
    }

    public static void main(String[] args) {
        //Get the number of arguments from the command line
        int numberOfArgs = args.length;
        int temp=MaxNodes;

        //Too many arguments
        if (numberOfArgs > 1) {
            System.out.println("Invalid number of arguments!");
            System.out.println("Syntax is:");
            System.out.println("BlackBoardTest [numberOfNodes]");
            System.out.println("Where numberOfNodes is the number of" +
                "nodes you want to test.");
            System.exit(-1);
        }
        try {
            if (numberOfArgs == 1) {
                temp = Integer.parseInt (args[0]);
            }
        }
        catch(NumberFormatException e){
            //bad argument use defualt value;
            System.out.println("bad argument format using default value of " +
                MaxNodes + "!");
            temp = 100;
        }
        MaxNodes = temp;
        //Lets run the gui instead of the text tester!
        junit.swingui.TestRunner.run(BlackBoardTest.class);

        //junit.textui.TestRunner.run(suite());
    }

    public static Test suite() {
        //return new TestSuite(BlackBoardTest.class);
        int i;
        //Run the tests
        TestSuite suite = new TestSuite();
        suite.addTest(new BlackBoardTest("testConstraints"));
        suite.addTest(new BlackBoardTest("testEmptyBlackBoard"));

        for(i=0; i < 3; i++) {
            suite.addTest(new BlackBoardTest("testHalfFullBlackBoard"));
            suite.addTest(new BlackBoardTest("testFullBlackBoard"));
            suite.addTest(new BlackBoardTest("testQuickDeleteBlackBoard"));
            suite.addTest(new BlackBoardTest("testHalfFullBlackBoard"));
            suite.addTest(new BlackBoardTest("testFullBlackBoard"));
            suite.addTest(new BlackBoardTest("testSlowDeleteBlackBoard"));
        }
        return suite;
    }

    /* Start writing my test suite */

    public void testConstraints() {
        int i;

        System.out.print("Testing constraints....");
        //Test the number of MaxNodes
        assertTrue(MaxNodes >=2);

        //Make sure maxNodes is even
        assertEquals(MaxNodes%2,0);
        System.out.println("Done!");
    }

    /*Test the empty blackboard*/
    public void testEmptyBlackBoard() {
        //create the blackboard
        BBMethods = new BlackBoard();

        //make sure its not null
        assertNotNull(BBMethods);

        //make sure there are no elements in it
        assertEquals(BBMethods.size(), 0);

    }

    /*Fill the blackboard with half the LanNodes we want*/
    public void testHalfFullBlackBoard() {
        String ip;
        int i;
        Random r;

        //Start the random number generator
        r = new Random();

        //Pick a random number of nodes to add between 0 and MaxNodes/2
        numNodesToAdd = Math.abs(r.nextInt()) % ((MaxNodes/2)+1);
        assertTrue(numNodesToAdd >=0 && numNodesToAdd <=(MaxNodes/2));
```

```
//Check if LanNode is ok
LanNode tempNode = new LanNode();
assertNotNull(tempNode);

//Fill the BlackBoard will the made up nodes
System.out.print("Adding " + numNodesToAdd + " ip addresses!....");
for (i = 0; i < numNodesToAdd; i++) {
    try {

        //Get a uniqe random IP address
        ip = getRandomIp();
        assertNotNull(ip);


        tempInet = InetAddress.getByName(ip);

        tempNode = new LanNode(tempInet);

        //Add the nodes in the blackboard
        assertTrue(BBMethods.add(tempNode));

    } catch (UnknownHostException e) {fail("Bad ip address added!");}

    tempInet = null;
}
System.out.println("Done!");
System.out.print("Testing newly added nodes...");
//Make sure size is ok
assertEquals(BBMethods.size(), numNodesToAdd);

//Change the states of the blackboard (all up and all down)
    tempNode = new LanNode();
    int numNodes = 0;

    try {
        /* Get up until either all nodes are checked out or max
         * number of threads reached */
        while ((tempNode != null) && (numNodes < MaxNodes)) {
            tempNode = BBMethods.getForUpdate();
            if (tempNode != null) {
                nodeArray[numNodes] = tempNode;
                r = new Random();

                //Get number between 0 and 1
                upArray[numNodes] = Math.abs(r.nextInt()) % 2;

                //need to wait for new random number
                try {Thread.currentThread().sleep(10);}
                catch(InterruptedException e) {/*Do Nothing*/}

                if (upArray[numNodes] == 1) {
                    tempNode.setIsUp(true);
                    tempNode.setSnmpEnabled(true);
                    tempNode.setServerStatus(true);
                    tempNode.setPolling(true);
                }
                else {assertEquals(0,upArray[numNodes]);
                    tempNode.setIsUp(false);
                    tempNode.setSnmpEnabled(true);
                    tempNode.setServerStatus(true);
                    tempNode.setPolling(true);
                }
```

```
                       numNodes++;
                   }
               }
           }
           catch(LanNodeIsDeleted e){fail("LanNode should not be deleted");}

           //tempnode should be null because we got all the nodes
           assertNull(tempNode);

           //the nodes we got out should be equal to the nodes we put in!
           assertEquals(numNodes,numNodesToAdd);

           assertEquals(numNodes, BBMethods.size());

           //Checkin all the nodes
           for(i=0; i < numNodes; i++) {
               BBMethods.checkIn(nodeArray[i]);
           }

           assertEquals(numNodes, BBMethods.size());

           //Checkout all the nodes and make sure all the values are ok

           //Check the states of the blackboard (all up and all down etc)
           tempNode = new LanNode();
           numNodes = 0;

           /* Get up until either all nodes are checked out or max
            * number of threads reached */
           while ((tempNode != null) && (numNodes < MaxNodes)) {
               tempNode = BBMethods.getForUpdate();
               if (tempNode != null) {
                   assertTrue((tempNode.isUp() == true) || (tempNode.isUp() == false));
                   numNodes++;
               }
           }

           //tempnode should be null because we got all the nodes
           assertNull(tempNode);

           //the nodes we got out should be equal to the nodes we put in!
           assertEquals(numNodes,numNodesToAdd);

           assertEquals(numNodes, BBMethods.size());

           //Checkin all the nodes
           for(i=0; i < numNodes; i++) {
               BBMethods.checkIn(nodeArray[i]);
           }

           assertEquals(numNodes, BBMethods.size());
           System.out.println("Done!");
}

public void testFullBlackBoard() {
   //fill the blackboard to the top!
   String ip;
   int i;
   Random r;

   //get the number of nodes needed to fill up the blackboard
   int numNodesToAddFinal;
```

```
numNodesToAddFinal = (MaxNodes - numNodesToAdd);
assertTrue(numNodesToAddFinal > 0);

//Check if LanNode is ok
LanNode tempNode = new LanNode();
assertNotNull(tempNode);

//Fill the BlackBoard will the made up nodes
System.out.print("Adding " + numNodesToAddFinal  + " ip addresses!....");
for (i = 0; i < numNodesToAddFinal; i++) {
        try {

            //Get a uniqe random IP address
            ip = getRandomIp();
            assertNotNull(ip);

            tempInet = InetAddress.getByName(ip);

            tempNode = new LanNode(tempInet);

            //Add the nodes in the blackboard
            assertTrue(BBMethods.add(tempNode));

        } catch (UnknownHostException e) {fail("Bad ip address added!");}

        tempInet = null;
    }
    System.out.println("Done!");
    System.out.print("Testing newly added nodes...");
    assertEquals(MaxNodes, BBMethods.size());

    //Change the states of the blackboard (all up and all down)
    tempNode = new LanNode();
    int numNodes = 0;

    try {
        /* Get up until either all nodes are checked out or max
         * number of threads reached */
        while ((tempNode != null) && (numNodes <= MaxNodes)) {
            tempNode = BBMethods.getForUpdate();
            if (tempNode != null) {
                nodeArray[numNodes] = tempNode;
                r = new Random();

                //Get number between 0 and 1
                upArray[numNodes] = Math.abs(r.nextInt()) % 2;

                //need to wait for new random number
                try {Thread.currentThread().sleep(10);}
                catch(InterruptedException e) {/*Do Nothing*/}

                if (upArray[numNodes] == 1) {
                    tempNode.setIsUp(true);
                    tempNode.setSnmpEnabled(true);
                    tempNode.setServerStatus(true);
                    tempNode.setPolling(true);
                }
                else {assertEquals(0,upArray[numNodes]);
                    tempNode.setIsUp(false);
                    tempNode.setSnmpEnabled(true);
                    tempNode.setServerStatus(true);
                    tempNode.setPolling(true);
                }
```

```
                   numNodes++;
               }
           }
       }
       catch(LanNodeIsDeleted e){fail("LanNode should not be deleted");}

       //tempnode should be null because we got all the nodes
       assertNull(tempNode);

       assertEquals(numNodes, BBMethods.size());

       //Checkin all the nodes
       for(i=0; i < numNodes; i++) {
           BBMethods.checkIn(nodeArray[i]);
       }

       assertEquals(numNodes, BBMethods.size());

       //Checkout all the nodes and make sure all the values are ok

           //Check the states of the blackboard (all up and all down etc)
           tempNode = new LanNode();
           numNodes = 0;

           /* Get up until either all nodes are checked out or max
            * number of threads reached */
           while ((tempNode != null) && (numNodes <= MaxNodes)) {
               tempNode = BBMethods.getForUpdate();
               if (tempNode != null) {
                   assertTrue((tempNode.isUp() == true) || (tempNode.isUp() == false));
                   numNodes++;
               }
           }

           //tempnode should be null because we got all the nodes
           assertNull(tempNode);

           assertEquals(numNodes, BBMethods.size());

           //Checkin all the nodes
           for(i=0; i < numNodes; i++) {
               BBMethods.checkIn(nodeArray[i]);
           }

           assertEquals(numNodes, BBMethods.size());
           System.out.println("Done!");
}

    public void testQuickDeleteBlackBoard() {
       //Delete entire blackboard at once
       System.out.print("Testing quick delete of all nodes....");
       LanNode tempNode = null;
       BBMethods.clear();
       assertEquals(BBMethods.size(),0);
       assertTrue(BBMethods.isEmpty());
       tempNode = BBMethods.getForUpdate();
       assertNull(tempNode);
       numip=0;
       System.out.println("Done!");
    }

   public void testSlowDeleteBlackBoard() {
       //Delete the LanNodes 1 by 1
```

```
        System.out.print("Testing single delete of all nodes....");
        int i;

        LanNode tempNode = null;

        //Get the current LanNode
        for (i=0; i < MaxNodes; i++) {
            assertEquals(MaxNodes-i, BBMethods.size());
            tempNode = BBMethods.getForUpdate();
            assertNotNull(tempNode);
            BBMethods.remove(tempNode);
            //Minus one for the removed node
            assertEquals(MaxNodes-i-1, BBMethods.size());
        }
        //make sure blackborad is completely deleted!
        tempNode = BBMethods.getForUpdate();
        assertNull(tempNode);
        assertEquals(BBMethods.size(), 0);
        assertTrue(BBMethods.isEmpty());
        System.out.println("Done!");
    }


    //Create a random unique ip address that does not exist already
    public String getRandomIp()
    {
        int i,first,second,third,fourth;
        String ip="";
        boolean Flag=false;
        Random r;

        i=first=second=third=fourth=0;

        while (!Flag) {

            //Create the random ip
            r = new Random();

            first = Math.abs(r.nextInt()) % 255;
            first++;
            second = Math.abs(r.nextInt()) % 256;
            third = Math.abs(r.nextInt()) % 256;
            fourth = Math.abs(r.nextInt()) % 256;

            //check if the ip exsits in the listing
            ip = first+"."+second+"."+third+"."+fourth;
            Flag = true;

            for (i = 0; i <= numip; i++) {
               //duplicate found get another ip
               if (ipArray[i]==null) {break;}
               if (ipArray[i].compareTo(ip)==0){Flag = false;}
            }
               //try {Thread.currentThread().sleep(10);}
               //catch(InterruptedException e) {/*Do Nothing*/}
        }

        ipArray[i] = ip;
        numip++;
        return(ip);
    }
}
```

Appendix E - Phase 3 Test Module Used

```
/**
 * PingTester.java
 *
 * Created on March 9, 2004, 11:32 AM
 *
 * @author  Paul Paszynski
 */
import blackboard.*;
import backend.*;
import java.net.*;

public class PingTester {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
       BlackBoard BBMethods = new BlackBoard();

        //change the number of threads here
        int NUM_THREADS=5;


        /*Note number of nodes must be exactly divisible by the number of therads*/

        //change the number of nodes to create here!
        int numOfNodes = 12;

        //add 3 special nodes
        numOfNodes = numOfNodes + 3;

        LanNode tempNode = new LanNode();

         int numNodes = 0;
         int tempCounter = 0;
         int nodeCount = 0;
         InetAddress tempInet;
         long t1, t0, dt;

        pingThread pingArray[] = new pingThread[NUM_THREADS];

        LanNode nodeArray[] = new LanNode[numOfNodes];

        for (tempCounter = 0; tempCounter < numOfNodes-3; tempCounter++) {

            try {
               System.out.println("Adding 24.150.63."+tempCounter);
                tempInet = InetAddress.getByName("24.150.63."+tempCounter);


                tempNode = new LanNode(tempInet);
                BBMethods.add(tempNode);

            } catch (UnknownHostException e) {}

            tempInet = null;
        }


        //Add 3 Special nodes (loop back, unreachable, invalid)
```

```
try {
    //loopback node!
    System.out.println("Adding specail node 127.0.0.1");
    tempInet = InetAddress.getByName("127.0.0.1");
    tempNode = new LanNode(tempInet);
    BBMethods.add(tempNode);

    //unreachable!
    System.out.println("Adding specail node 1.2.3.4");
    tempInet = InetAddress.getByName("1.2.3.4");
    tempNode = new LanNode(tempInet);
    BBMethods.add(tempNode);

    //invalid
    System.out.println("Adding specail node 0.0.0");
    tempInet = InetAddress.getByName("0.0.0.0");
    tempNode = new LanNode(tempInet);
    BBMethods.add(tempNode);


} catch (UnknownHostException e) {System.out.println(e);}

tempInet = null;


//make sure size of blackboard is correct!
System.out.println("Number of added nodes is:" + BBMethods.size());
assert(numOfNodes == BBMethods.size());


//Ping all the nodes using threads
System.out.println("");
System.out.println("Testing Threaded pinging");

//Start the timer!
t0 = System.currentTimeMillis();
while (nodeCount < numOfNodes) {

    tempNode = new LanNode();
    numNodes = 0;

    /* Get up until either all nodes are checked out or max number
     * of threads reached */
    while ((tempNode != null) && (numNodes < NUM_THREADS)) {
        tempNode = BBMethods.getForUpdate();
        System.out.print("Node Number " + nodeCount + " ");
        System.out.println("Pinging: " + tempNode.getIp().getHostAddress());

        if (tempNode != null) {
            nodeArray[numNodes] = tempNode;
            numNodes++;
            nodeCount++;
        }
    }

        //Make sure our program is running as expected
    assert((tempNode == null) || (numNodes <= NUM_THREADS));

    /* Start pinging all the nodes we checked out note:
     * (MAX NODES CHECKED OUT IS = NUM_THREADS) */

    for(tempCounter = 0; tempCounter < numNodes;  tempCounter++) {
        pingArray[tempCounter] = new pingThread(nodeArray[tempCounter]);
    }

    //Wait for all the threads to finish
    try {
        for(tempCounter=0; tempCounter < numNodes; tempCounter++) {
            pingArray[tempCounter].join();
        }
    }
    catch (InterruptedException e) {
        System.err.println("Warning thread Interrupted!");
    }
    //At this point all the ping threads are done

    //check for vaild responses from the special nodes
    for(tempCounter = 0; tempCounter < numNodes;  tempCounter++) {
        String ip;
        ip = nodeArray[tempCounter].getIp().getHostAddress();
        if (ip.startsWith("127.0.0.1")) {
            assert(nodeArray[tempCounter].isUp() == true);
        }
        if (ip.startsWith("1.2.3.4")) {
            assert(nodeArray[tempCounter].isUp() == false);
        }
        if (ip.startsWith("0.0.0.0")) {
            assert(nodeArray[tempCounter].isUp() == false);
        }
    }


    //destory all the ping threads
    for(tempCounter=0; tempCounter < numNodes; tempCounter++) {
        pingArray[tempCounter] = null;
    }

    //All node Threads done and updated now do a check in
    for(tempCounter=0; tempCounter < numNodes; tempCounter++) {
        BBMethods.checkIn(nodeArray[tempCounter]);
    }

}
//End the timer!
t1 = System.currentTimeMillis();

//Report the amount of time it took to ping 50 nodes using threads
assert(t1 >= t0);
dt = (t1 - t0)/1000;

System.out.println("All ping calls returned as expected!");
System.out.println("It took aproximatly " + dt + " seconds to ping "
    + numOfNodes + " nodes using threads");

System.out.println("");
System.out.println("Testing Iterative pinging");

//Start the timer!
t0 = System.currentTimeMillis();

tempNode = new LanNode();
nodeCount = 0;
numNodes = 0;
//Get all  nodes and ping them 1 by one
while ((tempNode != null) && (numNodes < BBMethods.size())) {
    tempNode = BBMethods.getForUpdate();
```

```
            if (tempNode != null) {
                nodeArray[numNodes] = tempNode;
                numNodes++;
                nodeCount++;
            }
        }

        try {
            int t = 0;
            for(tempCounter = 0; tempCounter < BBMethods.size();  tempCounter++) {
                String ip;
                ip = nodeArray[tempCounter].getIp().getHostAddress();

                //Do some checking!
                if (ip.startsWith("127.0.0.1")) {
                    assert(nodeArray[tempCounter].isUp() == true);
                }
                if (ip.startsWith("1.2.3.4")) {
                    assert(nodeArray[tempCounter].isUp() == false);
                }
                if (ip.startsWith("0.0.0.0")) {
                    assert(nodeArray[tempCounter].isUp() == false);
                }

                //Ping the current node
                System.out.print("Node Number " + t + " ");
                System.out.println("Pinging: " + ip);
                nodeArray[tempCounter].setIsUp(pingThread.pinger(ip));
               t++;
            }
        }
        catch(LanNodeIsDeleted e){/*Do Nothing*/}

        //All pinging done do a checkin
        for(tempCounter=0; tempCounter < numNodes; tempCounter++) {
            BBMethods.checkIn(nodeArray[tempCounter]);
        }
        //End the timer!
        t1 = System.currentTimeMillis();

        //Report the amount of time it took to ping 50 nodes using threads
        assert(t1 >= t0);
        dt = (t1 - t0)/1000;
        System.out.println("All ping calls returned as expected!");
        System.out.println("It took aproximatly " + dt + " seconds to ping "
            + numOfNodes + " nodes iterativly");
    }
}
```

Bibliography

A Simple Network Management Protocol (SNMP), May 1990. California: Hughes LAN Systems. Retrieved 28 October 2003 from the World Wide Web: http://www.ietf.org/rfc/rfc1157.txt?number=1157

A Visual Index to the Swing Components. Santa Clara: Sun Microsystems. Retrieved 22 October 2003 from the World Wide Web: http://java.sun.com/docs/books/tutorial/uiswing/components/components.html

Concise MIB Definitions, March 1991. California: Hughes LAN Systems. Retrieved 26 October 2003 from the World Wide Web: http://www.ietf.org/rfc/rfc1212.txt?number=1212

Glossary, 2001. O'Reilly & Associates, Inc. Retrieved 6 November 2003 from the World Wide Web: http://www.oreilly.com/catalog/debian/chapter/book/glossary.html

Glossary. Retrieved 6 November 2003 from the World Wide Web: http://philip.greenspun.com/panda/glossary.html

Harold, Elliotte Rusty. *Java Network Programming 2nd edition*. O'Reilly & Associates, 2000.

Holzner, Steven. *Java 2 Black Book*. Arizona: Coriolis Technology Press, 2001.

Holzner, Steve. *Java Black Book*. Scottsdale, Arizona: Coriolis Group, 2000.

Internetworking Technologies Handbook Second Edition. Mcmillan Technical Publishing, 1998.

Interoperability Clearinghouse Glossary of Terms, 2003. Alexandria, VA: ICH Architecture Resource Center. Retrieved 6 November 2003 from the World Wide Web: http://www.ichnet.org/glossary.htm

Java 2 Platform, Standard Edition (J2SE), 2003. Santa Clara: Sun Microsystems. Retrieved 1 November 2003 from the World Wide Web: http://java.sun.com/products/jdk/1.2/index.html

Java Network Programming FAQ, 27 April 2000. David Reilly. Retrieved 22 October 2003 from the World Wide Web: http://www.davidreilly.com/java/java_network_programming/

Java SNMP Package. Drexel University: Jonathan Sevy. Retrieved 16 October 2003 from the World Wide Web: http://edge.mcs.drexel.edu/GICL/people/sevy/snmp/snmp.html

Management Information Base for Network Management of TCP/IP-based internets: MIB-II, March 1991. California: Hughes LAN Systems. Retrieved 24 October 2003 from the World Wide Web: http://www.ietf.org/rfc/rfc1213.txt?number=1213

Palmer, Grant. *Java Event Handling*. Upper Saddle River, NJ: Prentice Hall, 2002.

Poehlman, Skip. Computer Science 3SE3 Lecture. Hamilton, Ontario: McMaster University, January 2003.

Project NAPS, 16 October 2003. Hamilton: Project NAPS Message Board. Retrieved 3 November 2003 from the World Wide Web: http://www.creativestudent.com/naps/index.html

Serialization in the Real World, 29 February 2000. Santa Clara: Sun Microsystems. Retrieved 14 October 2003 from the World Wide Web: http://developer.java.sun.com/developer/TechTips/2000/tt0229.html

Serkerinski, Emil., ed. Computer Science 3EA3 Lecture. Hamilton, Ontario: McMaster University, September 2002.

SNMP, 2003. About Inc. Retrieved on 6 November 2003 from the World Wide Web: http://compnetworking.about.com/library/glossary/bldef-snmp.htm

Structure and Identification of Management Information for TCP/IP-based Internets, May 1990. California: Hughes LAN Systems. Retrieved 26 October 2003 from the World Wide Web: http://www.ietf.org/rfc/rfc1155.txt?number=1155

Uszkay, Gordon J. Computer Science 4ZP6 Lecture. Hamilton, Ontario: McMaster University, October 2003.