State Discovery: The Power of Ping

Amber Van Wyck

CRPC-TR99811-S
August 1999

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted November 1999

State Discovery: The Power of Ping

Amber A. Van Wyk
avanwyk@cs.caltech.edu
http://www.cs.caltech.edu/~avanwyk
Center for Research on Parallel Computation
Summer Research Program in Parallel Computing for Undergraduate Women
California Insitute of Technology
Pasadena, CA 91125

August 13, 1999

1 Introduction

Many organizations, especially universities, find it difficult to monitor the state of the growing number of machines in open laboratories. System administrators may find it difficult to always be aware of the state of each machine. That state may include basic functionality, what software is installed on each machine, and the amount of memory in each machine. In open labs, whole machines or individual components (e.g., memory) can easily be stolen.

This paper presents an application that monitors a network of machines. It discusses the background needed to understand the project described, design considerations, implementation, and future work and ideas.

2 Background

2.1 Original Idea

This project originated from an idea California Institute of Technology graduate students, Eve Schooler and Mika Nyström, had tossed around for quite some time. Another student had his machine stolen out of his office. If it were reattached to the campus network, would it be possible to find the machine based on the unique hardware address of the Ethernet card?

A related idea was that Nyström had written a system administration application to monitor the well-being of most of the machines in the Computer Science labs. His application monitored if the machines were up or down by "pinging" them periodically. The basic idea was to keep the machines up and usable, but the application could also be used to detect when a machine was being or had been stolen or disabled.

The motivation for the project was to expand on this idea. Added capabilities could track down and kill remnant processes, track other capabilities and configurations of the machines besides just up vs. down, or monitor the amount of memory in each machine. Instead of the program being told exactly which machines to check and ignoring all others, it could interface to the Domain Name System (DNS) and dynamically find out which machines were registered with the name server to be sure it was monitoring all machines on the network.

2.2 Ping

The definition of ping has a somewhat confusing, but interesting past. It may have originally been an acronym for **P**acket **IN**ternet **G**roper. It may also have been named to match the submariners' term for the sound of a returned sonar pulse [PING].

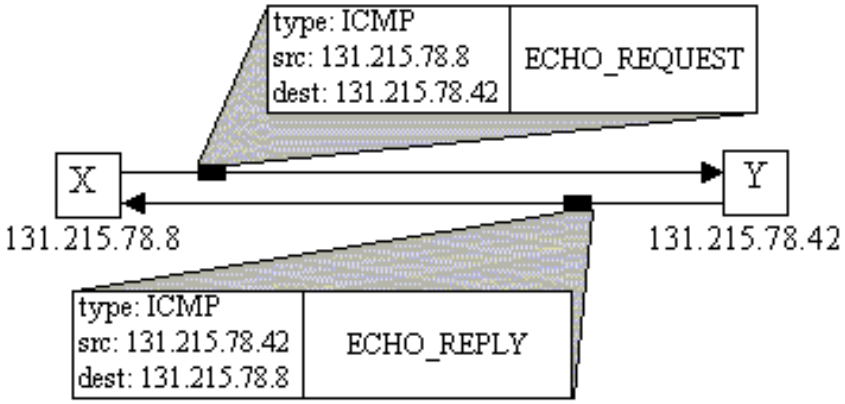

Figure 1. The Ping Process

The TCP/IP protocol suite requires that the Internet Control Message Protocol (ICMP) be implemented as part of the Internet Protocol (IP). ICMP supplies various means of ensuring messages are successfully transmitted. Two ICMP messages are ECHO_REQUEST and ECHO_REPLY. Applications that implement ping often utilize these messages to test whether a host is alive. It is a low-level way to check connectivity. In Figure 1, machine X sends an ECHO_REQUEST to machine Y. If Y is properly connected to the same network as X, Y will send an ECHO_REPLY to X.

Another method of implementing ping involves taking advantage of the echo service that Unix machines generally provide on

port 7. When this service is available, any message received at port 7 is echoed back to the sender. This provides another method of contacting a host to determine if it is alive.

In this paper, the term *ping* will be used to mean contacting a machine and using its response or lack thereof to ascertain if the machine is connected to the network.

2.3 Multicast

There are two main types of network messages. The first and most common is a unicast message. A unicast message is sent from one host to one other host. The second is multicast. A multicast message is sent from one host to a specific group of hosts on the network. Multicast address and group management are typically implemented in software, but the provision of hardware support is becoming more common. A machine has to explicitly join a multicast group to receive multicast messages sent to that group, however, a machine is not required to be in the group in order to send to it.

Internet Protocol v4 divides all Internet Protocol addresses (IP addresses) into 5 classes, lettered A-E [IP]. Each address is a 32-bit integer, which is most often converted into four dotted decimals. The first 1 to 4 bits of the 32 bit numbers specify in which class an address belongs. In Classes A, B, and C the addresses are divided into a prefix and a suffix. The prefix determines the network and the suffix determines the individual host being referenced. Figure 2 shows how the 32 bits are used for each class. Class E addresses are reserved for experimental use. Multicast addresses are Class D addresses. This means that the first four bits of the 32-bit addresses must be 1110. Therefore, the dotted decimal range for multicast addresses begins at 224.0.0.0 and ends at 239.255.255.255.

Class A: 0, 7-bit prefix, 24-bit suffix
Class B: 10, 14-bit prefix, 16-bit suffix
Class C: 110, 21-bit prefix, 8-bit suffix
Class D: 1110
Class E: 1111

Figure 2. IP v4 Addresses

There are multiple multicast addresses that are reserved or designated for specific services. For example, there is a reserved address that refers to "all routers on this subnet" and one reserved for SGI-Dogfight. This is coordinated by the Internet Assigned Number Authority (IANA). Another reserved multicast address that is particularly useful is 224.0.0.1. This address refers to "all systems on this subnet". The kernel automatically joins this group when a machine is booted [MCAST].

The most obvious advantage of contacting machines with multicast is the reduction of network traffic. Only one message has to be sent out to get N replies (Figure 3a), instead of N requests and N replies (Figure 3b). The machine on the far left needs to send a message to all of the other machines. Each arrow represents one message on the network. It is easy to see how the connections leading out from the sending machine would quickly become congested when using unicast. However, one problem encountered (in either case) is that each machine that responds to a ping sends its response back at virtually the same time. The network may get flooded, causing only a portion of the replies to get through to the ping server.

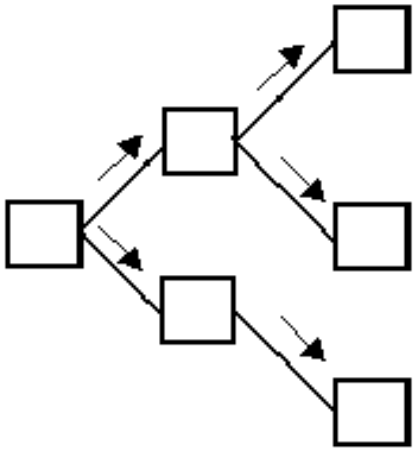

Figure 3a. Multicast

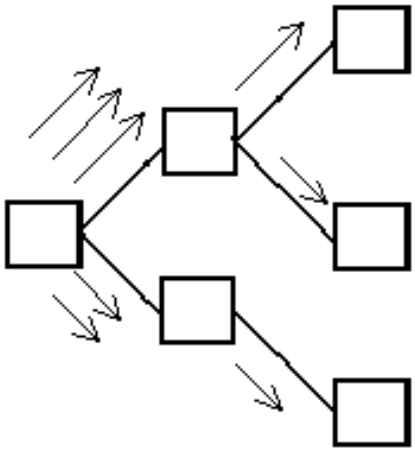

Figure 3b. Unicast

2.4 Architecture

There are two main issues to consider concerning architecture. There are centralized vs. distributed servers and active vs. passive communication. A centralized server architecture refers to the situation where one machine acts as the server and stores all of the information. Alternatively, in a distributed server architecture, N machines would perform the same functions as a central server. If a server is active, then it actively requests information from the clients. If a server is passive, then it patiently waits for the clients to send it information.

Nyström's original application uses active point-to-point communication with a centralized server architecture. This project utilizes active communication with a centralized server architecture as well.

3 Design Considerations

3.1 Language and Platform

Many programming languages and operating systems platforms were given consideration for this project. Languages considered were C, C++, and Java. Operating systems considered were Unix and Windows.

The following factors were considered:

- The original program was written in C.
- The programmer had minimal experience with C, but much experience with C++.
- The programmer had much experience with Java, including network programming.
- Java's only ability to ping relies on the port 7 echo service.
- Many machines do not provide a port 7 echo service.
- The real implementation of Unix ping (written in C) is over 1600 lines long.
- Java is better suited for the development of a Graphical User Interface (GUI).
- Unix has more utilities that may be helpful for such an application.

As a result, the program was implemented with Java under the Unix platform. It is capable of monitoring a heterogeneous network, but the server machine must support Java and Unix. It uses Javaâs *Runtime* class that gets the runtime environment of the current platform and allows the programmer to perform any task generally executed on the command line. In order for the program to be run on another platform these commands would need to be re-examined and possibly modified.

3.2 Implementation Plan

To extend Nyströmâs original program, the implementation plan focused on auto-configuration through DNS and multicast, as well as an improved presentation through a Graphical User Interface (GUI). The implementation plan is shown below:

- Explore the feasibility of using DNS to discover what machines reside on a given network.
- Use DNS to determine how the machines are organized and to collect their names and Internet Protocol Address.
- Devise a method for continuously pinging these machines to ensure their connectivity.
- Consider the possibilities of multicast
- Store and/or display the information collected.
- Consider other possibilities if time permits.

3.3 The Algorithm

A machine is considered *up* if it responds to a multicast or unicast ping. A machine is considered *down* if it does not respond to a two-ping series. The series could be two unicast pings or a multicast ping followed by a unicast ping. This is illustrated in Figure 4.

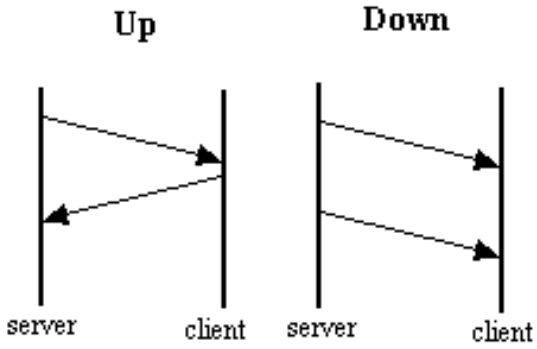

Figure 4. Up vs. Down

The following algorithm is used to track what machines reside on the network and to discover their current and ongoing status. An overview of it is depicted in Figure 5. The algorithm has three phases and behaves as follows:

Phase 1: DNS

- Use *host* as the interface to DNS to determine which machines reside on a given network. *host* is a program that is based on *nslookup* . *host* writes information to a file. The call to host is: *host öf filename öl domain.edu*
- Read the DNS information from the file, storing all machines by IP address(es). If an IP address is listed twice, the second name will be added to the record containing the same IP address. In this way, it allows for the case that a single machine will have multiple names, but it is not optimal in the situation where a single machine will have multiple IP addresses. Currently, if that is the case, that machine will be contacted once for each IP address. All machines are initially listed as using unicast communication and their states (*up* or *down*) are unknown.

Phase 2: Unicast or Multicast? Up or Down?

- Send out a multicast ping. Those that respond to this ping will be removed from the unicast list and placed in a multicast list. The state of these machines is set to reflect that they are using multicast. Any machine responding to multicast is presumed to be in an *up* state and will continue to be monitored with multicast.

- Ping each unicast machine and set the current state to up or down.
- All machines that have a down state are pinged again.
- If they do not respond a second time, a flag is set indicating not to ping them again and email is sent to inform the user of a state change.
- At this point, it has been determined which machines can be monitored with unicast, which machines are connected to the network and up, and which machines are down and thus presumably not connected to the network.

Phase 3: Iteration

- Sleep for a specified time, T_0.
- Send out a multicast ping. The program expects to receive a reply from everyone in the multicast list. If it does not receive a reply, those machines are treated as unicast machines that have missed one unicast ping.
- The program then pings all unicast machines that have an *up* state and sets their new states according to if a response is received.
- Ping machines that have a *down* state (did not respond to previous ping).
- If the machine is still *down*, the machine will not be pinged again and an email is sent to the administrative user informing her of the state change.
- Periodically, the list of multicast machines will be reselected.
- Periodically, each machine will be set to be pinged regardless of its previous activity.

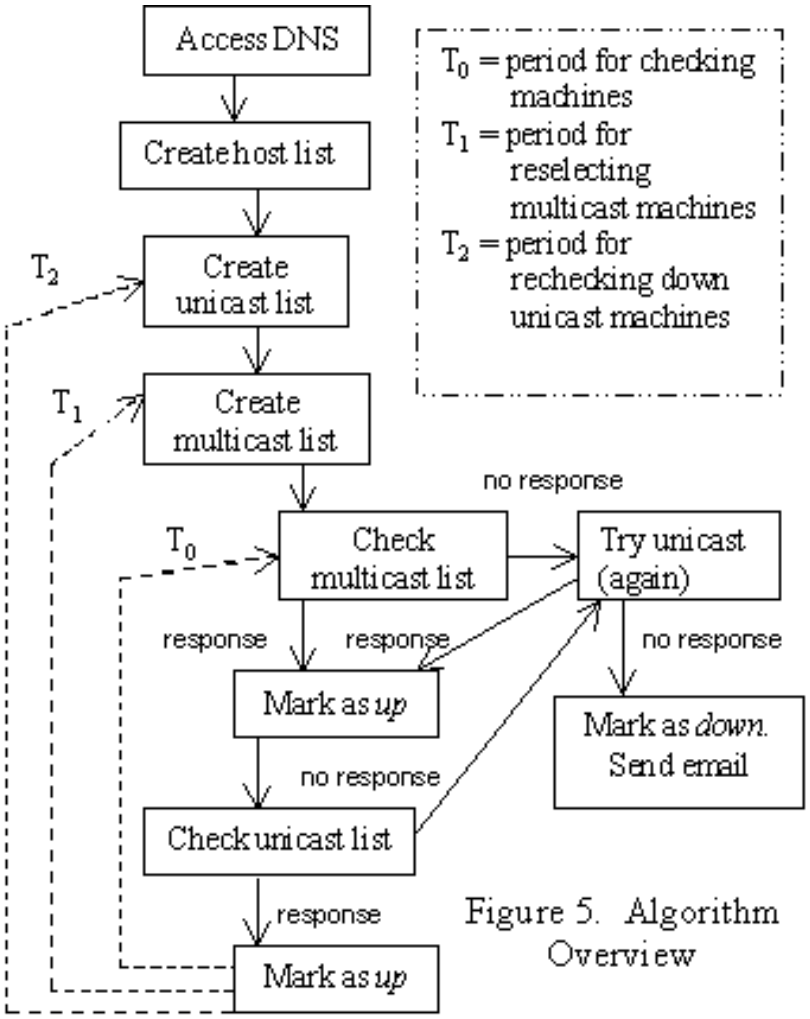

Figure 5. Algorithm Overview

The timing of the algorithm depends on the size of the network and the stability of machines connected to it. If a large number of machines are registered with DNS, but not connected to the network, then Phase 2 will take a considerable amount of time. It takes between ? and 3 seconds to determine that a machine is *up* and may take up to 12 seconds to determine that a machine is *down*. The time here will largely depend on the number of machines being contacted through unicast and the number of that set which is down. The timing for T_2 and T_3 are discussed below.

4 Implementation

There are three Java classes that implement the command line application. The *Machine* class stores information about each computer to be monitored. It includes set and get methods for most data members. The *PingThread* class is a *Runnable* interface that takes care of all of the processing to determine which machines are up or down and which are currently using multicast. The *NetMan* class contains the main method for executing the program.

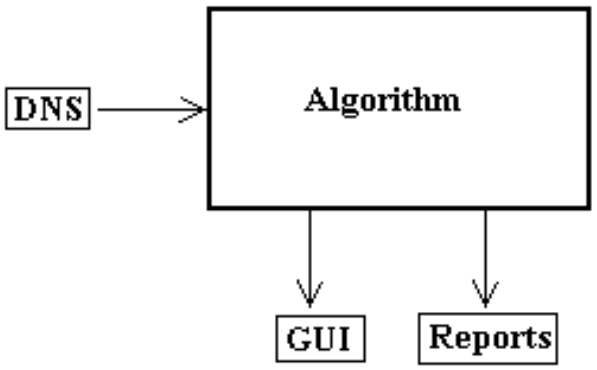

Figure 6. The Implementation

There are six options that can be specified when starting the program. The default amount of time for the program to wait between checking the machines is 60 seconds. This can be changed by *-s sleepsecs*. The default file name to put the host information in is *data.txt*. *-h hostfilename* will change that file. Reports are by default

created in *report.txt* . If a different name is desired it can be specified with *-r reportfilename* . When it is time for a new file to be created it is named with the month and day as a prefix. For example, a file created on August 2nd would be named *0802report.txt* , assuming the default file name. The user must specify if she wishes to use multicast with *-m* . The user must specify the email address where state change message are sent. This is done with *-e email* . An option that may only be useful to the programmer is *-d* . This prints vast amounts of debugging information. For example, if a user wishes to wait 2 minutes between checking all machines on *cs.caltech.edu* , to put the host information in *myhosts.txt* and the report information in *myreport.txt* , to use multicast, and to send state change messages to *avanwyk@cs.caltech.edu,* the command would be: *java NetMan -s 120 -h myhosts.txt -r myreport.txt -m thedomain.net*

The *NetMan* class takes the domain to monitor as a command-line argument, interfaces to DNS, creates the list of machines, and gives that information to *PingThread* for processing. There is no way to add a down machine to the list to be pinged except to wait until the next time that all machines are checked. When a machine officially goes down, an email is sent to a user informing her of the status change. Currently, the user to email is hardcoded, but this could easily be made into another switch for use on the command line.

It may be beneficial to periodically recheck every machine, including the ones that have been determined to be down (T_2 in Figure 5). Daily would be a good timeframe for performing this task. The program does not explicitly recheck every day at a specific time. It rechecks all machines after it has slept for the number of seconds in a day. It sleeps a specified time between each check of the *up* machines. So the actual time between rechecking all machines will be more than a 24 hour period since the processing time is not taken into account when deciding when to recheck. Also, when machines are rechecked a new file that contains the status data is started.

The same principle applies to the method of when to reselect multicast (T_1 in Figure 5). It reselects multicast four times "daily" using the same method as with rechecking all machines. Multicast machines are actually reselected after the thread has slept for one fourth of a day, not including processing time.

The program is currently set to use multicast only if the user is monitoring the local subnet because when it pings with multicast, it explicitly pings 224.0.0.1, i.e., all systems on the local subnet. This means multicast can only be used on the subnet on which the server machine resides. An error message is output and the execution stopped when this is not the case.

When a large number of machines is expected to respond to a multicast ping all responses may not be received due to message implosion. This means that the group of machines that consistently get their response back to the server machine may be small. If a machine's response to the multicast ping is not received, its state must be determined through the use of a unicast message. The fewer machines that are successfully monitored with multicast the less advantage there is to using it.

As with any program, it can be terminated with Ctrl-C or a *kill* command. However, the report file is not properly closed with this method and thus the current data is lost. This situation is remedied by having the program periodically check for input from the keyboard. If there is input and the input is a 'q', then the file is closed properly and the System.exit() method is used to terminate execution cleanly.

GUI

The Graphical User Interface version of the program works on the same basic premises as the command line version. There are more classes than the command line version as they are needed for implementing the windows used to create the graphical interface. The main class has been renamed Netgui, performing the same core services in addition to creating the graphical interface for the program. The command-line switches have been modified to be part of an options window as shown in Figure 7.

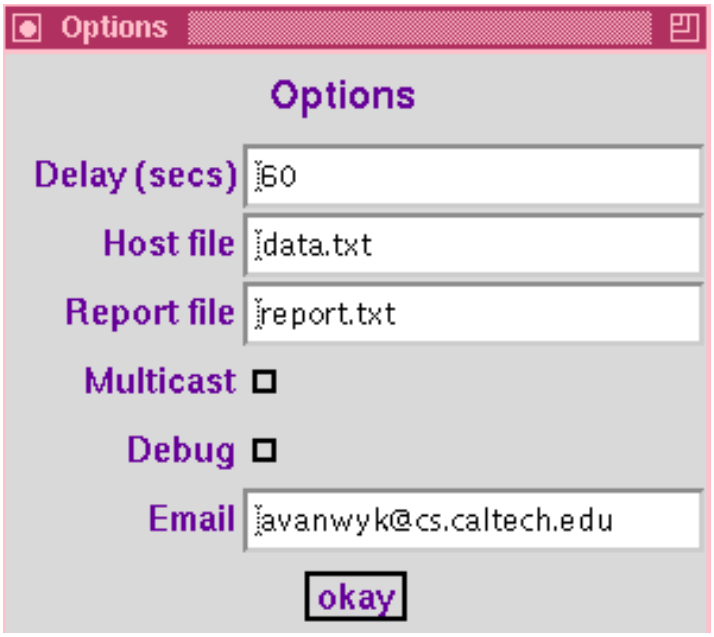

Figure 7. Options

After the program determines which machines are up, down, or multicasting, there are multiple options available to the user. These include viewing which machines are currently up, down, unicast, and multicast. A user can also view the general status of any machine, add a down machine back in to be rechecked, open a previous log file for viewing, and force a reselection of multicast machines or rechecking of all machines. A view of this form is shown in Figure 8. This screenshot was taken after the program had determined which machines fell into the four categories (up, down, unicast, multicast).

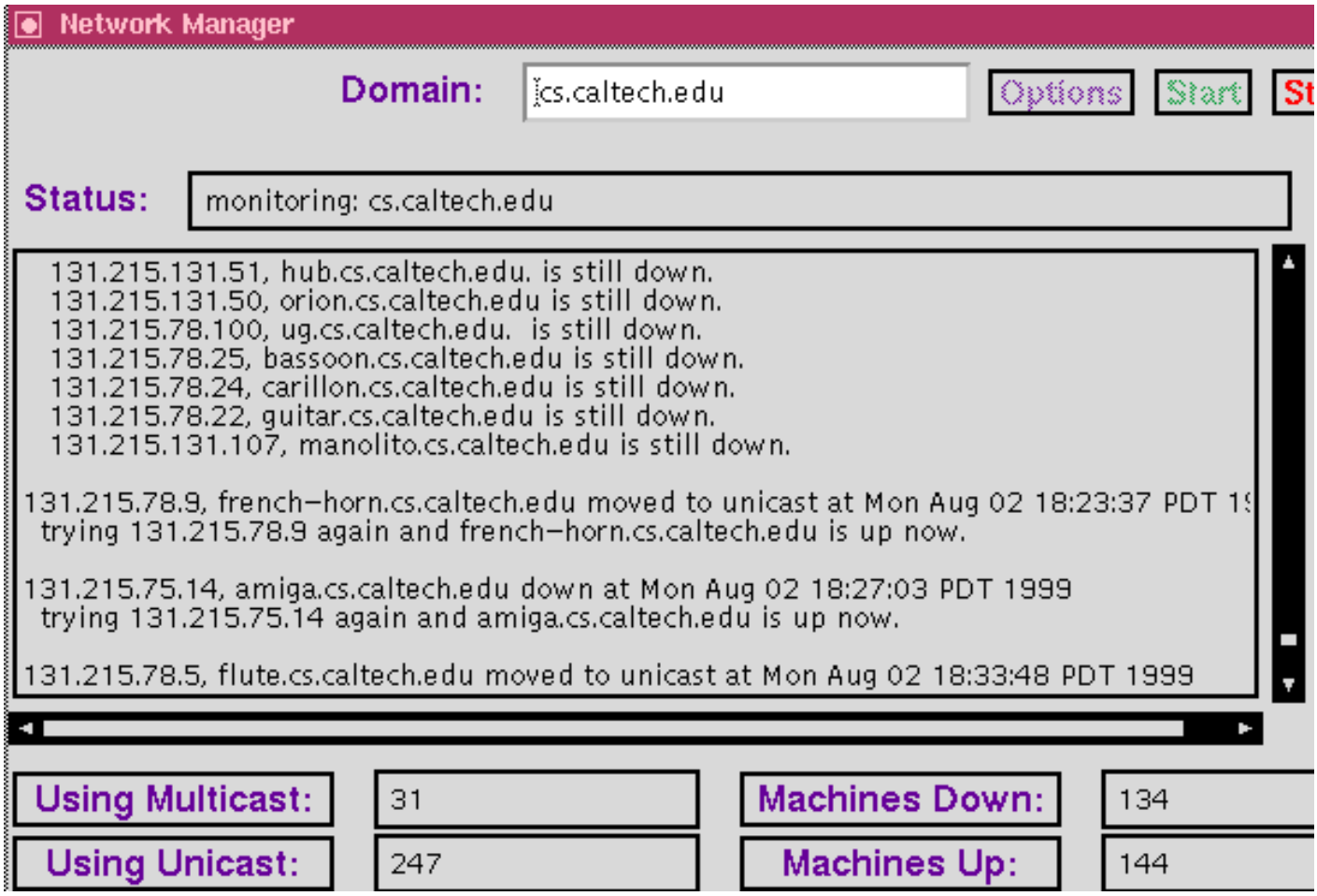

Figure 8. The Main Form

In this view, the cs.caltech.edu domain is being monitored. One hundred thirty-four machines are down while 144 are up. The high number of down machines may be due to the number of portable laptop computers that are registered but not currently attached to the network. Thirty-one machines are currently using multicast while 247 (of which 113 are presently up) are using unicast. The small number of multicast machines is due to the subdivision of the cs.caltech.edu domain into multiple subnets so only the subnet where the server sits will be able to utilize multicast in the current implementation. During some runs, the group of machines that consistently get their response through to the multicast ping is quite small. Other times, it has been quite large. (On the cs.caltech.edu subnet, the largest number of machines that responded to the multicast ping is 45 and the smallest is 36.)

The *Using Multicast* , *Using Unicast* , *Machines Down* , and *Machines Up* buttons bring up a pop-up windows that resemble Figure 9. These windows allow the user to browse through a list of machines currently in the corresponding state.

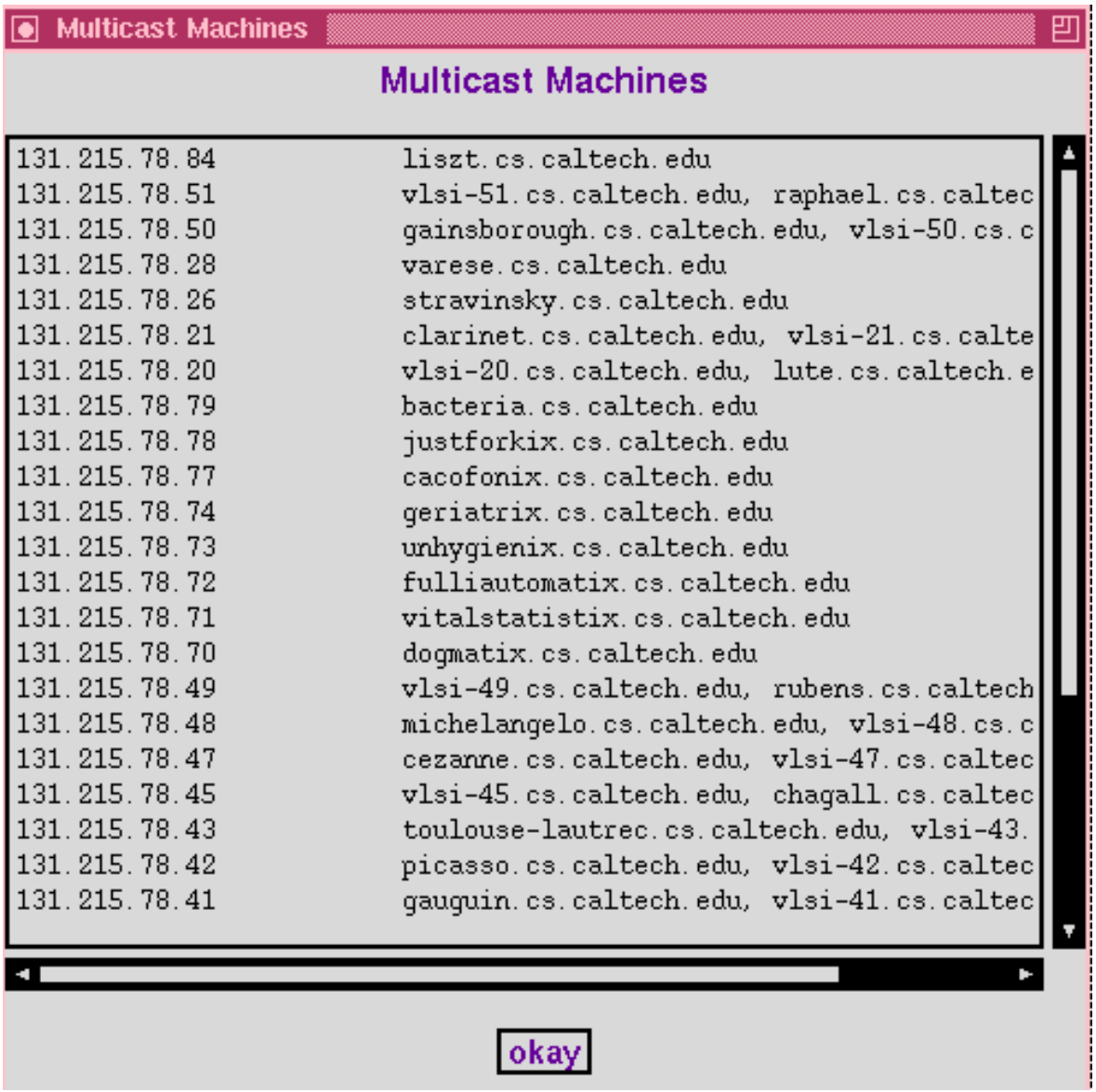

Figure 9. The pop-up window associated with the *Using Multicast* button

Currently the *Update/View Machine Status* button only allows the user to view a machine's name, address, and state information. This could be extended to allow the user to add such information as the physical location of a machine. In this case the set button would be needed to commit this information to memory or file. This pop-up window is shown in Figure 10.

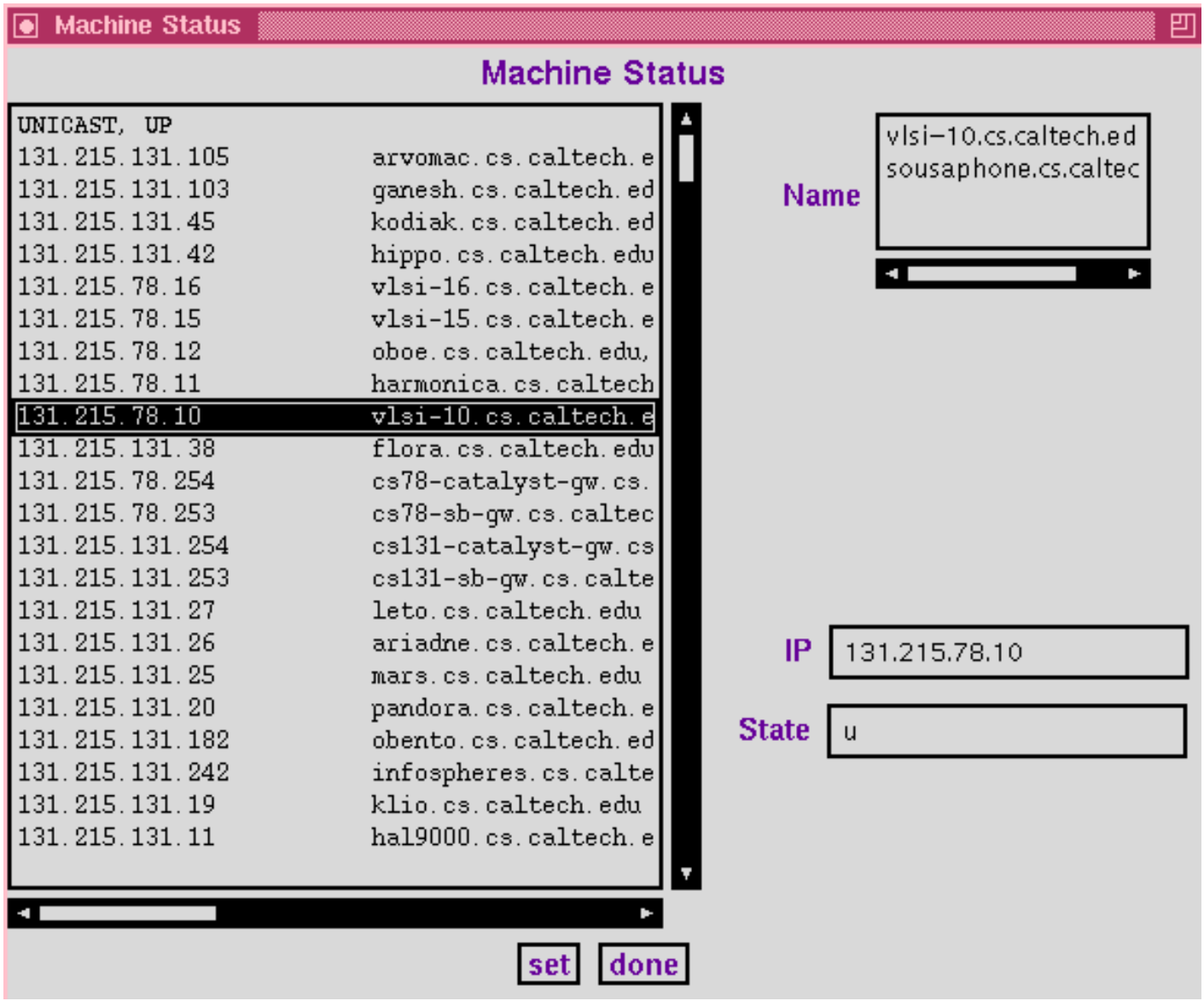

Figure 10. *Update/View Machin Status*

The *Add Machine to Check* button brings up a window that resembles Figure 9. In addition it contains an *add* button that allows the user to add a machine back in to be checked. Any number of machines can be added before continuing.

The *View Previous Logs* button would be best implemented if it brought up a file dialog box. This is possible by using the Java Foundation Classes. This is not implemented to date, and this feature currently asks for a file name and that file is opened by emacs. The user must know the name of the file, as browsing is not yet an option.

Future Work

An alternate architecture would use a distributed server. The network could be subdivided and one server would reside on each subdivision. Either communication technique could be utilized. The server could request information from its group or the server could wait for members of the group to offer information regarding their own state. If the server expected to hear from a machine and did not after a set period of time, it could then become active and attempt to initiate contact with that machine. An active-passive server combination would require a second application that would run on each machine being monitored.

Another centralized server approach would be to divide the monitored machines into multiple multicast groups. For example each client machine would join a specified multicast group. Then each machine would be expected to respond to one of N

multicast addresses. Figure 11 shows an example of multicast groups. Each cloud represents a group of machines that respond to a given address. For example, each group might represent a subnet. With this scheme, the server only needs to send out four messages to contact all of the machines. This design would allow the server to collect information from the other machines using strictly a multicast protocol, without the server subnet limitation.

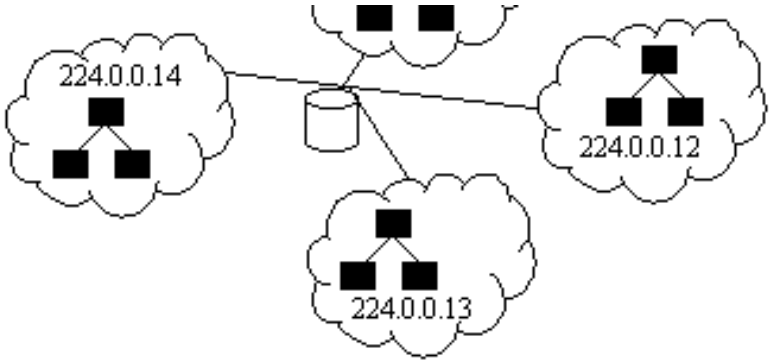

Figure 11. Multicast Groups

With a passive server, large multicast groups could be utilized by setting each client machine to wait a random time within a specified range before responding. For example, each machine would generate a random number and wait that long before responding. In this way some machines would respond immediately, and others would respond later. This would help to eliminate the problem of flooding the network with messages.

An alternate idea would be for the server to sniff the network to track which machines were sending internet packets. It would then build up a machine list based on detected activity. If any given machine did not send a message after a certain time period that machine would be pinged to make sure it was still connected. This would possibly cut down on network traffic since active machines would not be pinged. One drawback is that this scheme requires superuser or root privileges.

All of the information collected about each machine should be cached in a format that would allow it to be reloaded into the system in the event that the execution of the program was terminated. Any information that is manually input, such as physical machine location, would not have to be re-entered. Also, data concerning the long-term status of a machine could be collected.

Adding threads to collect other machine information such as memory, Ethernet address, and software installed is not yet possible, however this a feature that would be relatively simple to add. Another program could be written in any language and executed from within the Java program. In this way, the pinging could continue while other information was being collected and monitored in the background.

The original motivation for this project was to detect a stolen machine. However, the current program solves the broader problem of monitoring the machines in general. Finding a stolen machine could be an added capability, but is not the main focus of this application. This could be accomplished if the program was extended to collect and store Ethernet addresses as well as IP addresses.

Two other options that may be beneficial to add would be to let the user select the frequency of reselecting multicast machines and rechecking of all machines. Then the user could specify when to recheck these options. The easiest way to implement this would be to utilize the *Timer* class provided by the Java Foundation Classes (JFC or Swing). The Java Foundation Classes offer another set of classes for use in building user interfaces. However, they are not completely compatible with the Advanced Window Toolkit (AWT) and would thus require slight modifications across the entire GUI implementation. Another helpful feature would be the ability to specify times when a machine should be checked more intensively or not all in as in a case where an entire lab was closed and all its machines were shut off.

Conclusion

The presented application in command-line or GUI form can be used to track the up or down state of a set of dynamic machines on a network using multicast where possible. The application presented here is only a fraction of the possibilities for network monitoring, but we have listed many additional ideas in the future work section. Hopefully, time will permit an expansion of this application. Ongoing progress reports may be available at http://www.cs.caltech.edu/~avanwyk.

Acknowledgements

I would like to take this opportunity to thank the people who have made this work possible. I had a great experience this summer and I'm sure it is one that will continue to help me in the years to come. I would like to thank everyone at the Caltech Computer Science department for making this summer a memorable one. A special thanks goes out to graduate students Eve Schooler and Mika Nyström for their help and kindness. Another special thanks goes to JoAnn Boyd who did a great job of coordinating this program and taking care of us! Thanks!!

References

[IP] Stevens, W. Richard and Gary R. Wright. TCP/IP Illustrated, Volume 2 p. 155-156. 1995.

[MCAST] RFC 1700. http://www.rfc-editor.org

[PING] Editor Denis Howe, The Free On-Line Dictionary of Computing, http://foldoc.doc.ic.ac.uk