NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

IDENTIFYING ACCURATE RESOURCE MONITORING TOOLS AND TECHNIQUES

by

Ronald Jacobs Jr.

September 1999

Thesis Advisor: Debra Hensgen
Second Reader: Taylor Kidd

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1999	Master's Thesis

4. TITLE AND SUBTITLE: IDENTIFYING ACCURATE RESOURCE MONITORING TOOLS AND TECHNIQUES	5. FUNDING NUMBERS
6. AUTHOR(S) Jacobs, Ronald, Jr.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT:	12b. DISTRIBUTION CODE:
Approved for public release; distribution is unlimited.	

13. ABSTRACT *(maximum 200 words)*

Distributed applications concurrently share and compete for resources in heterogeneous systems. The objective of the Management System for Heterogeneous Networks (MSHN) is to use admission control, smart scheduling, and adaptation awareness in applications to successfully cope with the dynamics of resource availability. MSHN therefore requires knowledge of the expected resource utilization of applications that execute within the MSHN environment and the current state of these resources. MSHN relies on the above information to correctly identify resources to be assigned to these applications. This thesis investigates the capabilities of currently available communication resource status monitoring tools for the purpose of identifying those tools that, with low overhead, can provide accurate, end-to-end communication status information in a Windows NT environment.

The techniques used by the various tools are described and the methods for determining the accuracy of these tools are specified. Results of the experiments with the various tools show that they add between 2% - 3% overhead in most cases and as much as 10% overhead in the worst case. Finally, none of the existing commercial tools studied gave an accurate assessment of the end-to-end communication throughput and latency for Windows NT 4.0.

14. SUBJECT TERMS wrapper, passive monitoring, intercept system calls, resource monitoring, MSHN, heterogeneous computing, resource management system	15. NUMBER OF PAGES 84
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

IDENTIFYING ACCURATE RESOURCE MONITORING TOOLS AND TECHNIQUES

Ronald Jacobs Jr.
Major, United States Army
B.S., United States Military Academy, 1985

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1999

Author: _____
Ronald Jacobs Jr.

Approved by: _____
Debra Hensgen, Thesis Advisor

_____
Taylor Kidd, Second Reader

_____
Dan Boger, Chairman,
Computer Science Department

ABSTRACT

Distributed applications concurrently share and compete for resources in heterogeneous systems. The objective of the Management System for Heterogeneous Networks (MSHN) is to use admission control, smart scheduling, and adaptation awareness in applications to successfully cope with the dynamics of resource availability. MSHN therefore requires knowledge of the expected resource utilization of applications that execute within the MSHN environment and the current state of these resources. MSHN relies on the above information to correctly identify resources to be assigned to these applications. This thesis investigates the capabilities of currently available communication resource status monitoring tools for the purpose of identifying those tools that, with low overhead, can provide accurate, end-to-end communication status information in a Windows NT environment.

The techniques used by the various tools are described and the methods for determining the accuracy of these tools are specified. Results of the experiments with the various tools show that they add between 2% - 3% overhead in most cases and as much as 10% overhead in the worst case. Finally, none of the existing commercial tools studied gave an accurate assessment of the end-to-end communication throughput and latency for Windows NT 4.0.

TABLE OF CONTENTS

I. INTRODUCTION

Distributed applications concurrently share and compete for system resources and transmission bandwidth in heterogeneous systems. The objective of the Management System for Heterogeneous Networks (MSHN) is to use adaptation awareness in these applications to successfully cope with the dynamics of resource availability in heterogeneous networks. MSHN, a network-aware Resource Management System (RMS), monitors resource availability and use to deliver good end-to-end Quality of Service (QoS). Resource demands, based on past performance, are stored in a database. This information is then used to determine the computing resources required for an application and consequently, to locate the appropriate hosts for it. In addition to knowing the expected resource utilization of applications, MSHN must also know the current state of these resources. MSHN relies on the above information to correctly identify resources that must be assigned to these applications. This thesis investigates the capabilities of currently available communication resource status monitoring tools for the purpose of identifying those tools that can provide accurate, end-to-end communication status information in a Windows NT environment, with low overhead.

A. BACKGROUND

The major task of a heterogeneous distributed system is to provide a computing and communication platform to a set of disparate applications with different QoS requirements. A set of QoS requirements describes the quantitative needs of an application. These needs can be mapped directly to the resources contained within a distributed system. The QoS requirements are often expressed as a set of minimum QoS parameters, where each parameter describes a specific attribute of the distributed system. In other words, an attribute describes a single aspect of the functionality or the performance of an application within the heterogeneous environment. The QoS available from a heterogeneous system can be parameterized in several ways; some familiar

attributes include throughput, average end-to-end delay, the bound of the delay, loss ratio, and jitter.

The increasing demand for real-time applications to execute within a distributed system requires modifications to the existing network protocols to provide a better QoS. As a result, there has been a considerable amount of research aimed at enhancing existing network protocols to provide support for various QoS levels. These enhanced protocols must include a process to verify and maintain the level of quality that is requested specifically by each application. ATM protocols provide such QoS control ability. In an ATM network, when a connection is requested, the protocol takes into account the requests that it has already connected and the newly requested QoS in deciding whether to accept the connection. Once the connection is accepted, the requested QoS is provided as long as all connections are compliant with the traffic contract. Using newly proposed protocols in the frame relay network environment, customers at endpoints may also choose a certain class of service associated with a frame relay logical link [Ref.1]. These new protocols include Integrated Services (IS) and Differentiated Services (DS) and both will be discussed in detail later in this chapter. However, currently available packet-switched networks do not support similar QoS controls.

In a heterogeneous packet-switched network, the only supported QoS is reliability. For example, TCP/IP–the protocol suite most often used in the Internet–is designed to optimize overall network throughput, not to provide service according to each application's quantitative QoS requirement. A source does not notify the network of its QoS requirements before transmitting data to a destination and no resources in that network are set aside for any particular application. Therefore the actual QoS received by each Internet application–in terms of delay and throughput–varies over time. Real-time applications, such as video and voice, cannot tolerate the wide variations in delay and throughput present in today's heterogeneous environment. Generally, better services need to be provided on a per-application basis in order to satisfy the more strict QoS requirements of real-time applications, as well as to provide fair service to high priority, non-real-time applications.

Within a distributed system, a variety of applications share and compete for resources. These applications may require certain resources from the distributed system, but in times of large demand, the system cannot constantly supply these resources to the application. Here are a couple of reasons why. First, physical limitations may exist in the distributed environment; for example, a low bandwidth transmission path may be prone to errors during periods of high bandwidth variations. Second, dynamic activation and deactivation of multiple concurrent threads and processes within applications may cause variations within an environment without the proper reservation and assurance mechanisms. In any system without real-time prioritized scheduling and the reservation of Central Processing Unit (CPU) resources, CPU intense applications may not be able to receive a constantly high level of QoS with respect to a specific deadline requirement, a preferred application version, or the timeliness of the real-time application execution.

Consequently, it is essential for RMSs, such as MSHN, to schedule and manage applications, given the variations in resource availability, within the heterogeneous environment. RMSs help applications adapt to resource status changes within the heterogeneous environment. RMSs attempt to prioritize and allocate resources to applications so that they can meet their QoS demands. They account for variations in the status of resource availability. RMSs arbitrate access to a wide range of resources. RMSs can manage and schedule applications, accounting for the number of processors or processing nodes, also matching requirements with the status and capacity of the heterogeneous environment. For example, the number of processing nodes may be fixed but the RMS may dynamically determine the identity of these nodes for an application's QoS.

One primary class of applications that MSHN supports requires that mission-critical deadlines not be violated more than 5% of the time over any period. For these applications, the QoS guarantees are extremely important and the allocated resources must be readily available and consistent during program execution. A second class of applications that MSHN manages may have flexible QoS demands, being able to achieve minimum functionality when fewer resources are available. These versatile applications can accept and tolerate some resource scarcities. They are often willing to

sacrifice performance along certain quality axes in order to preserve the quality of other, more critical axes. For these applications, it is desirable to trade off less critical quality parameters for preserving quality assurance for critical parameters. Therefore, RMSs play an important role in the scheduling and allocation of resources when constant guarantees are not possible, when physical limitations are present, or when it is impractical to predict and reserve all desired QoS resources.

The next sections of this chapter will describe the overall goal of MSHN and present the detailed scope of this thesis. The final two sections will discuss the major contributions of this research and conclude with the organization of this thesis.

B. MSHN'S OVERALL GOAL

The goal of MSHN is to provide a computing environment that delivers each user's specified quality of service, subject to the following three attributes:

- resource availability,
- the application's priority, and
- the preference of each user for different forms of the requested service [Ref. 2].

MSHN's objective is to support real-time as well as non-real-time applications. It does so by providing a middleware layer that monitors perceived end-to-end QoS. Given a set of disparate jobs and a set of shared, heterogeneous resources, MSHN will determine where and when to run each job in a way that maximizes a metric that incorporates a collection of application-specific quality of service measures [Ref. 3]. The support provided by MSHN will be aided by the adaptation awareness of some applications being supported. These applications, however, need not be designed to be aware of resource availability or capable of coordinating with the dynamics of the heterogeneous environment. MSHN will consider end-user priorities, real-time criticality of applications, adaptivity of applications, preferences, and deadlines as it attempts to

deliver the best QoS. MSHN monitors and manages the resources of the heterogeneous environment in order to make wise allocation decisions. These decisions will provide the currently executing set of applications with the best overall QoS.

There are several important advantages of implementing a middleware layer to support QoS adaptable applications. Primarily, it will provide global support for multiple, concurrent applications. MSHN coordinates jobs with respect to adaptive behavior and eliminates conflicts for resources. Additionally, MSHN ensures that the adaptive system that results will not become unstable.

C. SCOPE OF THIS THESIS

The purpose of this thesis is to determine which of a set of currently available monitoring tools will best provide MSHN with an accurate assessment of end-to-end communication status. The key component to enable a real-time diagnosis of application and network performance includes a complete set of application and network monitoring tools with the ability to trigger what gets monitored and when. Several questions need to be explored pertaining to the QoS of real-time applications and heterogeneous networks.

- How do we determine the accuracy of a given tool to monitor end-to-end communication status?
- Which tools provide measurements that are useful in predicting runtime?
- What is the overhead of various, currently available tools?

Additionally, the overhead of each tool must be assessed, although determining how to measure the overhead will prove to be straightforward once we determine a way to answer the above questions. MSHN requires the gathering of resource usage information for applications that run within the MSHN system as well as resource status information within the scope of the MSHN scheduler. The MSHN scheduler uses this resource information to make scheduling decisions. The methods used to implement the gathering of this resource information are subject to three constraints:

- The implementation must not require any changes to the operating system.

    1. The ultimate goal is the widespread acceptance of MSHN. Many potential users are reluctant to allow modifications to their operating system.

    2. When routine operating system upgrades do occur, we do not want to have to redesign and redistribute our system to include the features or improvements of this new release.

    3. We do not want to risk compromising the security features of the operating system by changing the kernel.

    4. Source code of all operating system releases may not be available for modification.

- Modifications to the user's application code must be minimized and preferably will not be needed.

- The overhead imposed by the information gathering mechanism should not be excessive.

The main focus of this thesis is to determine which, if any, currently available end-to-end communication performance-monitoring tools would be appropriate for MSHN. We want to determine the best way to monitor the following end-to-end communication QoS parameters:

- throughput availability and

- delay bound or latency

An accurate measurement of these QoS parameters requires an end-to-end communication performance-monitoring tool that provides accurate and useful information. In our experiments, we analyzed the information provided by a few communication performance-monitoring tools, we tested the accuracy of their results, and we measured the overhead associated with the results.

D. MAJOR CONTRIBUTIONS OF THIS THESIS

This thesis analyzes several tools to determine whether they can be used to accurately predict the runtimes of typical jobs. The techniques used by the various tools are described and the methods for determining the accuracy of these tools are determined. Results of experiments with the various tools are analyzed.

E. ORGANIZATION

The next chapter of this thesis describes the various tools that were considered as possible end-to-end communication performance monitoring tools in MSHN. Chapter III discusses our experimental setup in detail and motivates the way we chose to measure the accuracy of the tools. Chapter IV analyzes the results of our experiments, summarizes our findings and suggests some future work.

II. RELATED WORK AND TECHNIQUES CONSIDERED

This chapter discusses research that is most related to the research performed for this thesis. Additionally it enumerates, and summarizes, several alternative techniques and tools that have been proposed for measuring the status of communication resources. We were not able to analyze the capability of each of these tools. Therefore this chapter motivates why we analyzed the tools and techniques that we did.

A. RELATED WORK

This section describes research that is the most closely related to ours. We have not located any published research that tries to assess the accuracy of tools that attempt to measure the end-to-end status of communication resources. We have learned that similar research is currently being conducted on different network monitoring tools at the University of Tennesee by Rich Wolski's group, although, as of today there are no publications or drafts available yet on that work. Additionally, they are focusing on a slightly different problem.

1. Broadcast Collective Communications Model

Every broadcast collective communication benchmark tries to measure the time required to complete a communication, from the first send to the last receive. Although this quantity is important, other properties are sometimes overlooked, such as local processing time and the overlap of computation processing time and communication processing time. The Lawrence Livermore National Laboratory (LLNL) uses a model of collective communications based on the LogP model that characterizes the significance of performance properties [Ref. 4]: The LogP model uses four parameters to capture point-to-point communications:

- The send overhead – the time that a processor requires to send a message.

- The receive overhead – the time that a processor spends receiving a message.
- The latency time – the time that a message actually spends in transit from source to destination.
- The gap – the minimum interval between consecutive sends and receives.

LLNL extended the LogP model with two *per processor* parameters to capture collective communications more accurately. The first parameter, the per processor overhead, is the time that each processor spends sending and receiving messages. The second parameter, the per processor gap, is the minimum interval of time between consecutive sends and receives at each processor.

This extended model does not attempt to measure the total time it takes to complete a communication. Instead, it measures the time it takes a processor to complete an individual task. The model uses the maximum time to complete an individual task as an estimate of end-to-end communication. This research demonstrated that measuring latency to individual tasks, as opposed to measuring the latency of the entire broadcast, produced an accurate broadcast benchmark that scales very well.

2. Predictive Application Performance Modeling

The Purdue University Network-Computing Hub (PUNCH) performance modeling system is a distributed system that utilizes local learning algorithms to learn the correlation between runtime input parameters and the corresponding runtime-specific resource requirements within a computational grid environment [Ref. 5]. PUNCH automatically extracts the runtime-specific values of administrator-specific input parameters from arguments and files supplied at runtime. PUNCH tailors its predictions from variations based on its current performance or from variations within the computing environment. PUNCH employs a two-level knowledge base that allows learning algorithms to exploit the resources contained within the domain. PUNCH utilized three instance-based learning algorithms that predicted CPU time and network data transfer time. The modeling system found that the nearest-neighbor algorithm outperformed the weighted-average algorithm and the locally weighted-polynomial regression algorithm in

predicting the resource usage. PUNCH is currently developing a way to predict memory and disk space usage within the computational grid environment.

3. Linear Models for Host Load Prediction

Carnegie Mellon University evaluated linear models for predicting the Digital Unix five-second host load average from 1 to 30 seconds into the future [Ref. 6]. They found that host load is consistently predictable from past behavior and that practical linear time series models are powerful load predictors. The idea behind using a linear time series model in load prediction is to treat the periodic samples of host load as a stochastic process that can be modeled as a linear filter. The filter variables can be estimated from the past observations of the periodic samples. If the variability of the samples results from the action of the filter, the variables can be used to estimate future host load predictions. This research concluded that for CPU bound tasks, the intuitive linear relationship between host load and execution time holds. Therefore, relatively simple predictive models can be used to estimate host load prediction.

4. Predicting the CPU Availability of Time-Shared Unix Systems

To utilize available resources efficiently, an application scheduler must make a prediction of the status and performance of those available resources. The University of Tennessee examines the problem of predicting available CPU performance on Unix timed-shared systems in order to develop dynamic application schedulers [Ref. 7]. Rich Wolski's group evaluates the accuracy with which CPU availability can be measured using a Unix load average, the Unix utility *vmstat*, and the Network Weather Service (NWS) CPU sensor.

Three performance measurement methodologies are used and the NWS CPU sensor monitors each methodology for current CPU availability. The first methodology uses the Unix load average metric that measures the average runtime queue length. The second methodology uses the *vmstat* utility. The *vmstat* utility provides periodic readings of CPU idle time, user time, and system time. The third methodology combines the load average metric and the vmstat utility with a small probe. The probe occupies the CPU for

a short period of time (1.5 seconds) and records the CPU availability. The method (load average or *vmstat*) that records the CPU availability closest to the value obtained by the probe is used to generate all measurements until the next probe is executed.

To determine the accuracy of the above methodologies, the measurements observed are compared to an independent CPU bound test process. The test process measures the percentage of CPU availability and that value is used to compare the results of the CPU availability percentages measured by the three methodologies. The research demonstrated that short and medium term predictions of available CPU performance are accurate enough for use in application schedulers.

5. Direct Queries for Discovering Network Resource Properties

The development and performance of distributed applications depends upon the availability of accurate predictions of network resource availability. Carnegie Mellon University compares bandwidth predictions from the Simple Network Management Protocol (SNMP) to traditional predictions based on application history [Ref. 8]. Executing an application using time series prediction techniques produced the application-based prediction. Applying the same techniques to the series of SNMP bandwidth availability measurements produced the SNMP-based prediction. The research demonstrated that SNMP is sufficient for obtaining the information needed to predict performance availability directly from the network. However, unlike the research described in this thesis, they do not attempt to measure or predict end-to-end status.

B. TECHNIQUES CONSIDERED

1. MSHN Wrapper

The MSHN Wrapper is responsible for directly monitoring resource usage, estimating resource availability, triggering the operation of other specific resource monitoring tools and utilities, and updating the Resource Status Server (RSS) and Resource Requirements Database (RDD). It gathers resource usage information for applications that run within the MSHN environment. The method used to monitor this

resource information is based upon intercepting system calls [Ref 2]. Prior to run-time, MSHN's client library is linked with the object code of the application. During run-time the client library gathers information concerning the application's resource utilization by intercepting calls made to the operating system. The client library measures the QoS given to the application by the resource used. The client library does not actively load resources. It monitors and collects resource status information by passively observing the performance of resources, in particular, when possible in ascertaining both throughput and latency. We did not measure the accuracy of this technique, but we did build a system that can and will be used to ascertain the accuracy of this method. However, we did measure the overhead of this technique.

2. Ping

Ping is a common network-troubleshooting tool used to test a network connection between two hosts. When a ping occurs, a multi-byte size message is sent from a source node to a destination node and the roundtrip time is recorded. Based on the number of bytes sent, this roundtrip time provides a single or multi-packet measurement of network throughput. Although a small number of bytes are forwarded during program execution, the ping application is defined as an active network-monitoring tool because it must load the network in order to provide a throughput measurement. Pinger, a program we used for testing purposes, allows a user to configure a list of Internet Protocol (IP) systems that the system will ping and the intervals at which those systems will be tested [Ref. 9].

3. Netperf

Netperf, a Network Performance Benchmark, is a client–server monitoring tool that is used to actively measure various aspects of networking performance. Its primary purpose is to actively measure end-to-end bulk data transfer time and request/response performance using a number of different protocols [Ref. 10]. The bulk data transfer or "unidirectional stream" measures the speed at which a source node can send data to a destination node or the speed at which the destination node can receive data. Data is sent for a specified period of time and after the time has elapsed, measurements are taken to

determine the amount of data sent and received. Communication performance is measured in transactions-per-second. Netperf's request/response performance test estimates end-to-end latency by sending a short message from a local node to a remote node. The remote node responds to the message immediately and the local node records the roundtrip time. This roundtrip time is divided in half and the result is used to estimate latency. Since Netperf puts a load on the network when testing bulk data transfer and request/response performance it falls into the category of an active network-monitoring tool. Netperf was not used in our tests because the evaluation copy measured network performance for only ten seconds.

4. Gloperf

Gloperf, the Globus Network Performance Measurement Tool, performs periodic network performance tests between pairs of computers with different IP addresses using a "librarized" version of the Netperf utility [Ref. 11]. Using a Netperf library allows a single process to act as both a Netperf Client and a Netperf server. Gloperf uses a Netperf "TCP_STREAM" test to measure network throughput between hosts. Since the test uses TCP, the measurement assesses end-to-end data throughput of applications that use TCP/IP. Gloperf uses Netperf's "TCP_RR" test to measure latency. The latency value reported by Gloperf is the multiplicative inverse of the transactions-per-second value produced by the Netperf test. Like the throughput test, this measurement includes TCP/IP overhead and, like Netperf, Gloperf is also an active monitoring tool. Gloperf was not used in our testing because there is currently no version that executes in a Windows NT environment.

5. SNMP

In Schnaidt's research, Simple Network Management Protocol (SNMP) was rejected as a possible monitoring tool for MSHN because it provided link-based information and did not estimate end-to-end network throughput nor latency between machines on nonadjacent networks [Ref. 12]. It was hypothesized that any tool built on top of SNMP would have to be modified once routing algorithms were changed.

Nonetheless, SNMP is a simple request/response protocol that communicates management information between two types of SNMP software entities: SNMP Agents and SNMP Managers.

SNMP agents are software or firmware within a network device, such as a router or a repeater. The agent monitors a device's operation but does nothing unless it detects an error or is polled for information. These agents may also be installed on workstations to collect and report information such as available hard disk space, memory usage, and status of active applications. A device's agent is normally accessed using a Management Information Base (MIB) which contains device specific information that facilitates the operation of the management software and the agent.

SNMP managers are sophisticated software applications that collect and process resource information from many agents. SNMP managers may poll certain agents for resource information at regular intervals or they may poll on the request of a user. SNMP managers run in a network management station and issue queries to gather information about the status, configuration, and performance of external network devices or elements.

SNMP agents respond to network management station queries by sending unsolicited reports back to the network management station when certain network activity occurs. For security reasons, the SNMP agent validates each request from an SNMP manager by verifying its membership in an SNMP community and by verifying that it has the proper access privileges.

An SNMP community represents a logical relationship between an SNMP agent and one or more SNMP managers. The community has a name, and all members of a community have the same access privileges. The access privileges are read-only or read-write. Read-only privileges permit members to view configuration and performance information. Read-write privileges permit members to view configuration and performance information or modify the current configuration. SNMP agents only respond to requests from those managers that are members of one of its communities. If the agent authenticates the community name in the SNMP message and can authenticate that the manager generating the request is a member of that community, it gives the requested access allowed for members of that community. Thus, the purpose of the

SNMP community is to prevent unauthorized managers from viewing or changing the configuration of any network device.

SNMP is an active and passive monitoring tool because of its nature as a response/request protocol. Managers are active; they query agents, they can reconfigure agents, and they listen for reports from agents. Agents are passive most of the time because they respond to queries, they maintain data specific to devices, and they generate problem reports only when necessary. Many tools have been built on top of SNMP to permit users, applications, and RMSs to easily determine resource status. SNMP was not used for this testing but it will be used in future tests.

6. Remos

A Resource Monitoring System for Network-Aware Applications (Remos) is a query-based interface that monitors the status network of resources [Ref. 13]. Queries can be made to Remos to determine the structure of the network environment or to obtain information about specific sets of nodes and communication links in the network. Remos is built on top of SNMP and a Remos specific collector is used to translate Remos queries into SNMP queries. The main features of the Remos interface include the following:

- Logical network topology: Remos supports queries about the network's characteristics from an application's standpoint, which may be different from the physical network topology.
- Flow-based queries: Queries regarding bandwidth and latency are supported for logical communication channels between nodes. Flows and flow-based queries represent application-level connections.
- Multiple flow types: Remos supports queries relating to fixed flows, variable flows, and independent flows.
- Simultaneous queries: Applications can make queries about various flows concurrently. Remos will account for any resource sharing by these flows.
- Variable time-scale queries: Queries are not dependent on size, allotted time, nor availability of future resources.

- Statistical measures: Remos reports all quantities as estimates. Dynamic measurements typically display heterogeneity that does not always correspond to a known distribution.

The Remos Application Programming Interface (API) is divided into three classes of functions: status functions, fitting functions, and topology queries. Status functions return information on compute intense nodes and source node to destination node flows. Fitting functions return information on the possibility of a network supporting several simultaneous flows. Fitting functions also permit applications to determine the communication service that will be rendered by a new set of flows, given the existing flows communication resource requirements' and the shared properties of the network. Topology queries obtain a network topology for a set of nodes selected by the user executing a required application. This topology represents the network's performance characteristics as seen by the application. Topology and flow queries support the need for global decision making and allocation of resources. Remos focuses on providing support for application-level access to shared network information. However, because Remos uses SNMP as the underlying resource monitoring agent, it is considered an active monitoring tool. Remos was not used in our testing because there is currently no version that executes in a Windows NT environment.

7. NWS

The Network Weather Service (NWS) is a distributed service that dynamically forecasts the performance of various networked resources in regards to the applications that use those resources [Ref. 14]. NWS uses a distributed set of software sensors from which it gathers readings of instantaneous conditions. It couples these instantaneous readings with numerical models to generate forecasts of future conditions for a requested time frame. NWS aims to maximize four, possibly conflicting, functional characteristics. It tries to meet these goals despite the highly dynamic execution environment and evolving software infrastructure within heterogeneous systems. The goals are:

- Predictive Accuracy: NWS must be able to provide accurate estimations of future resource performance in a timely manner. As stated earlier in this

chapter, the designers of NWS are currently researching how well their techniques do in achieving this goal.

- Non-intrusiveness: The system must load the resources it is monitoring as little as possible.

- Execution longevity: NWS must be available at anytime as a general system service. It should not execute and complete – its execution lifetime is logically indefinite.

- Ubiquity: As a system service, NWS should be available from all potential execution sites within a resource set. Similarly, it should be able to monitor and forecast the performance of all available resources.

NWS network sensors rely on active network probes when determining network load. Each probe consists of a timed network operation, such as the movement of a fixed amount of data, or, in the case of TCP, the establishment and dissolution of a network connection. At regular intervals, each network sensor connects to a set of peer sensors running on machines of interest. These sensors conduct one or more probes of different types to gather their measurements. To gather a set of end-to-end performance measurements of any type from n sensors requires $n^2 - n$ probes. To avoid introducing a heavy network load, sensors are organized hierarchically so that an end-to-end measurement can be made for a representative subset of the total sensor population.

Currently, the NWS network sensor attempts to measure three network performance characteristics: small message roundtrip time, large message throughput, and TCP socket connect and disconnect time. The small message probe consists of a 4-byte TCP socket transfer that is timed as it is sent from a source sensor to a destination sensor and back. The socket connection used to facilitate the transfer is established before the probe is conducted. Large message throughput, which measures available network bandwidth at the application level, is calculated by timing the transfer of a message using TCP and the acknowledgement of the receipt by the receiving sensor. The size of the message, the sending and receiving socket buffer requests, and the size of the internal buffers used by each sensor in the socket can be set to different values for each sensor.

NWS attempts to measure end-to-end network performance between all possible network sensor pairs by continually operating each sensor. Therefore, NWS is an active monitoring tool and an all-to-all sensor communication consumes a considerable amount of network and host machine resources. NWS alleviates this problem somewhat by creating a hierarchy of sensors sets named cliques, whereby each sensor operating in a clique conducts inter-machine communications with other clique members within that set. However, the sensor population needs to be monitored extensively to keep the size of the clique from disrupting the overall network QoS.

8. Netlogger

The Distributed Application, Host, and Network Logger (Netlogger) is a set of tools that does performance and bottleneck analysis on distributed systems [Ref. 15]. Netlogger monitors the behavior of all the elements of the end-to-end network communication path in order to determine resource utilization. It determines whether hardware components require upgrading in order to alleviate future QoS problems. Netlogger creates event logs, a representation of raw information about system performance using the Netlogger Toolkit, a tool that generates and manipulates the logs. Netlogger monitors events using the Unix utilities *netstat* and *vmstat. Netstat* is a tool that reports the contents and statistics of the network i.e., throughput, latency and overall performance. *Vmstat* reports on the host machine i.e., configuration and load, virtual memory, disk, and CPU activity.

Within a heterogeneous environment, Netlogger modifies distributed application components as well as some operating system components in order to perform precision time stamping and event logging at critical points in time. Netlogger is a somewhat passive approach to monitoring resources within a distributed system. The events are correlated with system behavior in order to characterize the overall QoS of the system and network during actual operation. The monitoring is designed to facilitate identification of bottlenecks, performance tuning, and network performance research. It also estimates the measurement of throughput and latency characteristics for distributed loads. Software agents collect and filter event-based performance information, turn on or

off various monitoring options to adapt the monitoring to the current system state, and manage the large amounts of log data that are generated. The goal of this performance characterization work is to produce high-speed components that can be used as building blocks for high-performance applications. This method may also provide an information source for applications that can adapt to component congestion problems. Netlogger was not used in our testing because there is currently no version that executes in a Windows NT environment.

9. WinDump

Windump is the Win32 porting of one of the most used UNIX network capture and analysis programs: TCPDump [Ref. 16]. WinDump allows users to capture and view packets that are passing on a network. WinDump was written to work with Ethernet networks, but can be used without problems on other networks as well. WinDump uses the packet capture library (LIBPCAP) for the capture process. This library, written originally for UNIX, implements a set of high level capture functions. WinDump also uses the Network Driver Interface Specification (NDIS) packet capture driver, a device driver that interacts with NDIS to capture the packets from the network. WinDump was used in our testing.

10. Protocol Analyzer

Protocol analyzers capture conversations between two or more systems or devices. They not only capture the traffic, but they also decode and interpret the traffic. Decoding allows users to view the "conversation" in English, as opposed to binary language. Protocol analyzers also provide statistics and trend information on the captured traffic. They provide information about the traffic flow on any LAN where device-specific information can be viewed. Unlike SNMP-based management consoles, protocol analyzers are device independent.

Observer, Network Instrument's Local Area Network (LAN) Analyzer and Troubleshooting Tool, is a protocol analyzer that monitors network traffic [Ref. 17]. It

contains active and passive monitoring techniques that provides three main sources of information: network statistics, packet capture and decode, and trending information.

Network statistics are provided about traffic flow, station health and network or station line errors. This information helps identify trends and general conditions that may provide end-to-end QoS analysis, may signal an unexpected network problem or condition, or may locate a load issue that is causing slowdowns. Packet capture and decode displays LAN traffic, or packets, decoded into specific functions and sub-functions for problem isolation. Viewing the specific packet-by-packet conversion can show exactly what is happening during a system-to-system communication, both when things are functioning correctly and when things are not. Trending information displays historical usage data over days, weeks, months or even years. This information provides an historical perspective on network resource utilization, problems within the network, and the trending information may be used to identify a potential problem before it happens.

Observer uses protocol statistics to display the percentage bandwidth that a particular protocol is using. This would help a human (slow) user determine QoS requirements, efficient segmentation, and allows for problem isolation based on application or server type. Observer uses station statistics to show the traffic generation by each station, server, bridge, router and the percent of the total bandwidth each station is using. With this information, you can determine who is using bandwidth and what stations or devices are using more bandwidth than expected. For example, if one station is sending 40% of the total data sent this could indicate either a faulty network adapter, execution of multiple retries, or simply a device that consumes more network bandwidth than expected. In either case, having a protocol analyzer allows QoS information to be captured that can be used to construct appropriate resource utilization predictions based on facts, not guesswork. Observer's packet capture and decode captures traffic in real time, records it, and presents the decoded information. Packet decodes show conversations between source nodes and destination nodes. This information helps in any problem situation by showing exactly what is happening and when, and exactly which device is doing what. Observer was used in our testing.

C. CHARACTERISTICS OF THESE SYSTEMS

1. Active vs Passive

The tools we described in this chapter can be categorized as either active or passive monitoring tools. Active monitoring tools are independent of the operating system and distributed applications but they add supplementary traffic to the network that could impact negatively on the end-to-end communication QoS. When accurate performance measurements are needed, active monitoring tools could nullify the accuracy of these measurements by adding traffic to the network. Also, if too much traffic is added, the active monitoring tools could increase the network load well beyond the transfer capacity of the network. Conversely, passive monitoring tools do not place additional loads on the network but previous ones have been tailored to a particular operating system and distributed application.

2. Clock Synchronization

Some of the related work we described above uses timestamps and requires a clock synchronization protocol to make accurate measurements. The synchronization protocol determines the time offset of a server clock relative to a client clock. On request, the server sends a message including its current clock value or timestamp. The client records its own timestamp upon arrival of the message and this communication continues until the client eventually synchronizes to the server. For the best accuracy, the client needs to measure the server-client propagation delay to determine its clock offset relative to the server. Since it is not possible to determine the one-way delays, unless the actual clock offset is known, the protocol measures the total roundtrip delay and assumes the propagation times are equal in each direction. All host clocks within a distributed system can be synchronized to within a millisecond of each other [Ref. 15].

D. SUMMARY

In our review of network monitoring tools and components, we classified each as active or passive. Active monitoring tools are independent of the operating system and the distributed application but they must load the network in order to obtain measurements of network resources. Passive monitoring tools have the ability of monitoring end-to-end communication resources without putting an additional load on the network. All tools require a clock synchronization protocol to obtain accurate measurements. The next chapter describes our approach to testing some of the active and passive tools for accuracy and overhead.

III. APPROACH AND EXPERIMENTAL SETUP

MSHN must be able to obtain accurate information about the availability and status of communication resources. It is essential for MSHN to obtain accurate measurements about the communication resources' capabilities and status in order to allocate the necessary resources to distributed applications that require a certain QoS. To avoid dependencies on the idiosyncrasies of network architectures and communication systems, MSHN requires a technique that accurately measures availability of these resources. This thesis investigates both active and passive techniques. Ultimately, a monitoring technique that is independent of the OS, programming language, network technology and topology, and hardware is preferred.

This chapter describes the model and methods used to investigate the accuracy of information provided by communication resource monitoring tools. First we address the challenges that a resource-monitoring tool must overcome to provide accurate measurements. The next section identifies the categories of methods that are used for measuring the availability of communication resources. Then we define the system parameters that influence the performance of applications. Following that we present the technique that we used to measure the accuracy and the model that was used as the base-line for our experiments. Finally we describe how the experiments were conducted.

A. RESOURCE MONITORING CHALLENGES

Recall from Chapter I, the three constraints imposed upon MSHN's technique for assessing the status of resources: (1) the implementation must not require any changes to the operating system; (2) modifications to any application's code must be minimized; and (3) the overhead imposed by the information gathering mechanism should not be excessive. In this section we briefly summarize the challenges a resource-monitoring technique must address while conforming to the above constraints.

1. Accuracy

The information provided by a resource-monitoring technique in regards to heterogeneous networks and distributed applications needs to be accurate enough to provide a useful estimate of resource availability. If the estimates are not accurate enough, then the system may assign resources that will not satisfy the required QoS constraints. Furthermore, the estimates must be of high enough fidelity, therefore they must be based on an adequate number of sample measurements taken at an appropriate time.

2. Sharing

Application level connections between source and destination nodes share physical resources with other application level connections. This dynamic sharing of resources is the major reason for the variable communication performance experienced by applications. Parallel and distributed applications often simultaneously transfer data across multiple point-to-point connections, which also leads to sharing, as these connections compete with each other for resources.

3. Information Diversity

Applications may require a wide variety of information, ranging from static network topology and dynamic bandwidth estimations on a variety of times scales, to latency information. This QoS information may have to be retrieved through diverse mechanisms not available in any single monitoring technique.

4. Network Heterogeneity

Network architectures differ significantly in their configurations. Some configurations lack the ability to collect and report network information such as communication performance, link capacities and utilization, and network topology. The nature of the information that is made available by any network architecture impacts the design, overhead, and accuracy of information associated with a monitoring technique.

In addition, most techniques report only about network resources and do not report about other resources that impact the end-to-end communication QoS.

5. Abstraction Level

This challenge addresses two concerns: which communication resources should be monitored and at what level the monitoring should occur. One solution is to monitor as many resources as possible, at a very low level. However, monitoring low level or system specific information conflicts with other goals, particularly the portability of the resource information across heterogeneous networks. For example, cell information provided by an ATM protocol would be meaningless in a packet-switched environment. Monitoring low level or system specific resources also raises ease of use and scalability concerns because most networks are too extensive to monitor. The passive or active techniques employed to gather such fine-grained information might provide a huge volume of data, from which it would be difficult to extract a concise description of resource availability. An alternative is to provide the resource information at a much higher level. Moreover, a high level of abstraction can make the resource monitoring less intensive and may avoid information overload, but it can also result in less accurate information.

6. Dynamic Behavior

Network conditions can change quickly, so the monitoring technique must be able to adapt to a dynamic environment. Furthermore, different distributed applications may request slightly different types of QoS, e.g., some applications are more interested in burst bandwidth while others require long term average bandwidth. Additionally, applications are most interested in future network behavior, not historical information, and hence it is important to be able to use this historical information to predict future behavior.

While all of these resource-monitoring challenges are important, this thesis is mainly concerned with the accuracy of the monitoring technique and the end-to-end shared resources within a heterogeneous framework.

B. NETWORK QUALITY OF SERVICE METHODS

This section discusses three methods that use communication bandwidth to determine end-to-end status of communication resources. All three methods rely on a time series prediction model, which uses a series of measurements to make predictions of future behavior. The difference between the three methods lies in the measurements that are taken and the accuracy of those measurements.

1. Application Methods

One method for measuring the status of communication resources is by actually running an application that uses resources and observing the performance obtained from those resources. However, some disadvantages of this method are that the application may not use the resources whose status needs to be determined and when the application is run with different parameters, different resources will be used. The client library defined for MSHN [Ref. 2] is an example of this approach.

2. Benchmark Methods

Benchmark methods use a small set of representative applications, called benchmarks or probes, to monitor the communication resources. Examples of benchmark methods include SNMP [Ref. 14], NetLogger [Ref. 15], NetPerf [Ref. 10], and Pinger [Ref. 9]. However, the problem with using benchmarks for monitoring communication resources is that the load that these benchmarks place on the resources may need to be excessive in order to obtain accurate values [Ref. 3]. Benchmarks, an active monitoring technique, also may give inaccurate measurements of the communication resource availability because it is also measuring the load that it has placed on the network.

3. Network Methods

Rather than obtaining end-to-end resource communication status, another approach is to approximate it using the status of the network only. We term such approaches as *network methods*. An example of a network method is a protocol analyzer.

Network methods offer several improvements over the use of benchmarking or application monitoring methods:

- Direct measurement of the network status gives absolute knowledge of the network resources, including the amount of traffic, the latency of individual messages, error rates, utilization history, and current throughput at any instance or over any period in time.

- Direct network snooping and filtering imposes significantly lower loads on the network and other resources than active benchmarks, because it uses its own resources.

Although network methods impose less of a load on the resources than the methods above, they can only be used to approximate the end-to-end communication status. However, this approximation may suffice if there is another method for obtaining information about local communication resources with which this approximation can be combined. Additionally, network methods require special hardware. This disadvantage is offset by the fact that this hardware can often be queried remotely.

C. APPLICATION CHARACTERISTICS

This section describes the characteristics of applications that influence resource availability. The applications we are interested in supporting may have all or any of the following four characteristics. First, they may be interactive; such tasks are initiated or guided by a user that desires a certain responsiveness and predictable behavior. Second, applications may display elasticity in the case of missed deadlines; such delays do not make these applications unusable, but merely result in a lowered QoS. The third characteristic that applications may contain is distributed operations; in this case applications are implemented in a disbursed manner. Finally, applications may be characterized by their adaptability; adaptable applications expose controls, called application specific QoS parameters, that can be adjusted to change the amount of computation and communication resources a task requires.

Applications, which contain one or more of the above characteristics, require a QoS that is suitable to the application and the network environment. Although most of today's applications require local, remote, and network resources to function, there are some applications that utilize one type of resource more than another. Applications that utilize more local and remote CPU resources are classified as compute intensive applications, while applications that utilize more network resources are classified as communication intensive applications. The remainder of this section will discuss compute and communication intense applications and the factors that influence the performance of those types of applications.

1. Compute Intensive Applications

Compute intensive applications utilize mainly local and remote CPU resources. The performance of computation intense applications are mostly influenced by these factors:

- The short term scheduling algorithm used by the OS.
- The speed of and shared load on the CPU.
- The language(s) of the application and the chosen compiler and/or interpreter.
- The application's input parameters.

These factors most definitely affect the performance of computational intense applications and they must be considered when trying to predict runtime and the end-to-end QoS of such applications. Additionally they also affect communication intense applications, but not in such an obvious way.

2. Communication Intense Applications

Communication intense applications utilize the network resources more than they utilize CPU resources. The performance of communication intense applications are mostly influenced by the following factors:

- The average number of messages transmitted.
- The average size of each message transmitted.
- The total bandwidth of the network.
- The average, peak, and lower bound of available network throughput.
- The current network latency.

These factors most definitely affect the performance of communication intensive applications and they must be considered when trying to predict their performance. As stated above, however, the factors that affect computationally intense applications also affect communication intensive applications. Therefore, MSHN is interested primarily in predicting and ascertaining end-to-end latency rather than network latency and available end-to-end bandwidth, rather than network bandwidth.

D. EXPERIMENT SETUP

Much recent research has focused on accurately measuring and predicting the availability of a machine's CPU [Ref. 4, Ref. 5, Ref. 7]. One of the hardest problems facing these researchers so far has been to determine how to assess, quantitatively, which techniques did the best job of measuring this availability. This thesis focuses on accurately measuring the availability of the end-to-end communication resources and one of the hardest problems that we faced was also to determine how to assess which techniques provided the most accurate assessment of the availability of these resources.

1. Assessing the Accuracy of End-to-End Communication Resource Availability Measurement Tools

The accuracy of various techniques must be compared against some known, unquestionable, quantity. We initially considered using techniques that were supported by their own hardware, such as protocol analyzers and packet sniffers, but decided against it because such techniques only measure the network resource. As we stated in the last section, there are many factors such as short term scheduling algorithm and

choice of compiler that affect the performance of all programs, not just computation-intensive programs [Ref. 3]. We also considered comparing commercial and experimental techniques against MSHN's client library because, according to our reasoning, the client library had been designed to accurately measure the end-to-end communication resource availability. We rejected this approach as well because we wanted to also quantitatively assess the accuracy of this library to measure this availability. We finally settled upon executing communication-intense applications, in an environment where they could have exclusive access to all resources, and recording the wall-clock time required to execute the applications. In order to assess the accuracy of the measurements from each technique, we would execute the technique concurrently with the application to try to obtain the technique's approximation of the communication resource availability (end-to-end throughput and latency). If possible, we would then use the technique's estimate, together with formulas described in Carff's thesis [Ref. 18], to predict the completion time of the application. Then, the most accurate technique would be the technique whose predictions most closely matched the actual runtime of the application.

Executing the various techniques within an autonomous domain will also give us an estimate of the overhead associated with each technique. First, we execute the communication intense application alone, several times, so as to acquire a statistically valid set of runtimes. Second, we execute the same communication intense application concurrently with one of the chosen techniques, again, numerous times, so as to produce another statistically valid set of runtimes. Third, we compare the set of runtimes with the technique to the set of runtimes without the technique to identify the technique's estimated overhead. These experiments use an isolated network. Therefore, any difference between runtimes will be directly due to the technique.

2. The Experimental Application

The baseline application used in the experiments, called an emulator, emulates applications consisting of multiple processes that communicate with one another. The original design of the emulator, implemented by Paul Carff, permitted the simultaneous

use of compute and communication resources [Ref. 18]. Computations were performed and messages were sent periodically. However, the class of applications used in our experiments requires the use of communication resources without computations in order to obtain reasonable estimates of runtime.

a. *Emulator Modifications*

We modified Carff's emulator for our experiments as follows:

- Deleted the compute intense thread. In deleting this thread, we also deleted the code that input the mean, standard deviation, and distribution describing the amount of computation that was to take place in between message send events.
- Deleted the code that synchronized the communication intense thread with the compute intense thread.
- Made all communication processes asynchronous *only*.

As in Carff's implementation, the modified emulator supports n applications that communicate with each other during their execution. The resulting emulator applications have a small computation portion that performs very minor calculations and a communication intensive portion that communicates with the other n - 1 applications.

The emulation system consists of a master process and some number of application emulators. Within each emulated application there are three threads, a main thread, a receiving thread, and a sending thread. The following provides some detailed characteristics of the emulation system:

- The emulator's master process reads the input parameters. Using these parameters, it executes the application emulators in every possible configuration. For example, if the input parameters specify two emulators and two machines, it executes four configurations: (a) both on first machine; (b) both on second machine; (c) the first process on the first machine and the second process on the second machine; and (d) the second process on the first machine and the first process on the

second machine. When each configuration terminates, the elapsed runtime is output.

1. This process determines the number of emulated applications that will be created.

2. It also starts all of the emulated applications and waits for them to complete.

- Each emulated application has both a sending and a receiving thread.

  1. The application's sending thread sends data using TCP.

  2. The receiving thread receives the data. It terminates when a sentinel is received.

- A clock-server is used to synchronize the clocks of all the machines to the closest microsecond.

b. ***Incorporates Functionality of MSHN Client Library***

We also modified the above emulator to include the functions of MSHN's client library. The client library, as discussed in Chapter II, was written in the C language and executes within a UNIX environment. The emulator was written in the Java language and although it can be executed in a Unix environment, we would need to modify the JVM, something that is not permitted by the standard license. Alternatively, we considered wrapping the system calls within the Java Virtual Machine (JVM), similar to the way that we wrapped the C library, but that was also not permitted by the standard license. Therefore, we chose to incorporate the client library's functionality within Carff's emulator. Our wrapped version of the emulator executes within a Windows NT environment and has the following characteristics:

- We measured the average end-to-end latency for each configuration. We did not have to identify the early-reader, late-writer scenario because we wanted the end-to-end latency that could have included time in the ready queue [Ref. 2].

1. In the sending thread, we wrapped every outgoing message with a header and we recorded the amount of time it took to send the entire message.

2. In the receiving thread, we inspected every incoming message for the header and we recorded the amount of time it took to receive the entire message.

3. The time difference where the sending thread stopped sending and the receiving thread started receiving is the end-to-end latency time.

- We measured the average throughput for each configuration. This can be used to test the accuracy of the tools we tested.

  1. We know the number and size of messages transmitted from sender to receiver.

  2. We know the total time, from the start of the first send to the end of the last receive, it took to transmit the messages.

  3. The number of messages multiplied by the size of messages gives us the total amount of data, in bytes, transmitted between sender and receiver. We divide that total by the amount of time it took to transmit the message and that number is the throughput.

3. Test Environment of the Experiment

Monitoring and finding the status of communication resources is a very large problem so we restricted our test environment for experimentation purposes. We used three distinct groups of three machines, each networked together on distinct 100BaseT Ethernet LANs. Within each group we monitored the communication resources with different monitoring tools to ascertain the accuracy of the tools' measurements. The test environment has the following attributes:

- Three distinct sets of machines containing single Pentium III 500Mhz processors and identical components (according to the manufacturer's specification) with Microsoft Windows NT Workstation 4.0 (NT 4.0) as the operating system.

  1. Each set of machines is connected by a common LAN.

2. The LAN is autonomous.

- The input parameters to this emulator are:

1. The number of machines (3).

2. The number of processes (3).

3. The names or addresses of the machines.

4. The mean number of messages sent and received between processes. (See Table 3.1 for values used)

5. The mean size of messages sent and received between processes. (See Table 3.1 for values used)

Table 3.1 shows the number and sizes of messages sent in our experiments. The computational resources needed to generate the actual size and number of messages were minimal; this generation never required more than 16ms.

Number of Messages Sent	Size of Messages Sent
250	8k bytes, 16k bytes, and 32k bytes
2500	8k bytes, 16k bytes, and 32k bytes

Table 3.1. Number and Size of Messages used in Experiments

No applications, other than the emulator and a clock server[1], were executing in the test environment.

[1] Several tools needed the clock server. Since the server synchronized at most every minute, it did not add much overhead.

E. WHAT THE TECHNIQUES SHOULD MEASURE

The technique that MSHN uses should assess the current end-to-end status of communication resources. This end-to-end status should include the following:

- Time required for a sender to prepare a message for transport.
- Time required for propagating the message.
- Time required for the receiver to process a received message.

Additionally, it should include system overhead, such as time that the process spent in the ready queue. If the technique provides an accurate estimate of the total time to perform the steps above, then we can use Carff's formulas to accurately predict the total runtime.

F. TECHNIQUES TESTED

We used three resource-monitoring techniques to assess the end-to-end status of communication resources. Two monitoring techniques are passive. The other monitoring technique is active. In addition to testing a modified version of the MSHN Wrapper, we tested WinDump [Ref. 16], a protocol analyzer, and the active technique, Pinger [Ref. 9].

G. SUMMARY

This chapter described the model and methods used to investigate the accuracy of information provided by communication resource monitoring techniques. It addressed the challenges that a resource-monitoring tool must overcome to provide accurate measurements and described the different methods used for network performance prediction. We defined the parameters that influence resource utilization in applications

and we presented the model that we used as the base line for our experiments. The next chapter presents the results of our experiments.

IV. RESULTS, SUMMARY, AND FUTURE WORK

In this chapter we present results from comparing various monitoring techniques. Three of the techniques that we used were passive and one was active. We measured the overhead of each and show in this chapter that the overhead is minimal for most techniques examined. In fact, all except WinDump had less than 2% overhead; WinDump had approximately 10% overhead. We also learned that only one of the techniques has the potential for predicting the end-to-end throughputs and latencies required by Carff's formulas [Ref. 18].

This chapter is organized as follows. The first section reviews our experiments and analyzes the overhead of each technique. The next section discusses the type of information that each of the techniques provide. After that, we discuss future work and explain how we built the foundation for that future work. Finally, we summarize the contributions of this research.

A. EXPERIMENTS AND OVERHEAD

As stated in the previous chapter, we performed our experiments in three separate domains, where each domain contained three identical machines. Each machine executes a server process that makes it possible for processes running on all three machines to communicate with one another. Within this three machine configuration, there are 27 possible schedules. We run each of the 27 schedules 30 times in order to get a statistically valid number of runtimes. A total of 810 schedules are executed, recorded, averaged and plotted for each experiment. Our goal is to compare the runtimes of the various tools when run concurrently with the emulator to the runtime result of the emulator. In other words, we want to assess the overhead added by each of the monitoring techniques as it executes concurrently with the emulator. Table 4.1 lists the 27 schedule numbers and the machine assignments of each schedule that was executed by the emulator. For example, schedule number 8 has a machine assignment of 231. This

means that process 1 is executed on machine 2, process 2 is executed on machine 3, and process 3 is executed on machine 1.

Schedule Number	Process # 1	Process # 2	Process # 3
1	1	1	1
2	2	1	1
3	3	1	1
4	1	2	1
5	2	2	1
6	3	2	1
7	1	3	1
8	2	3	1
9	3	3	1
10	1	1	2
11	2	1	2
12	3	1	2
13	1	2	2
14	2	2	2
15	3	2	2
16	1	3	2
17	2	3	2
18	3	3	2
19	1	1	3
20	2	1	3
21	3	1	3
22	1	2	3
23	2	2	3
24	3	2	3
25	1	3	3
26	2	3	3
27	3	3	3

Table 4.1. Mapping of Schedule Numbers to Machine Assignments

Recall from Chapter III that each process has a sending thread and a receiving thread. During one set of experiments, the sending thread and the receiving thread exchanged 40 MB of data per message. Figure 4.1 shows the runtimes, averaged over the 30 experiments, for the emulator when executed concurrently with each monitoring technique and when executed alone.

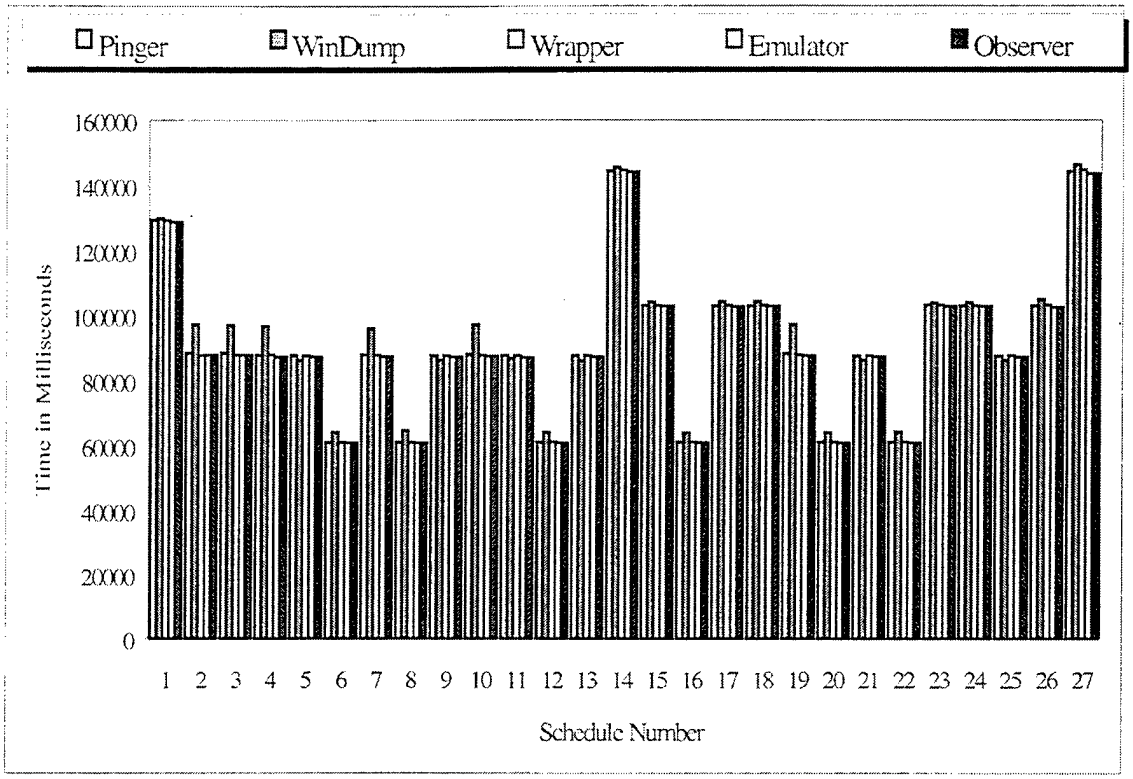

Figure 4.1. Comparisons of Monitoring Technique Runtimes

We note that the actual runtimes of the emulator are just slightly lower than the runtimes for the Pinger and the Wrapper application. Moreover, the worst runtime occurrence for the Pinger application is within three percent of the emulator's runtime and the worst runtime occurrence for the Wrapper application is within two percent of the emulator's runtime. We did execute the Wrapper application with the message numbers and sizes that were presented in Chapter III, Table 3.1. The Wrapper runtimes were still within two percent of the emulator runtimes. Therefore, the overall communication overhead experienced by applications executed concurrently with the Pinger and Wrapper is negligible.

We want to also note that the worst runtime occurrence for the WinDump application is approximately ten percent higher than the emulator's runtime. However, the overhead observed from the WinDump application is not due to the utilization of communication resources. The overhead is attributed to the use of the CPU as it outputs data to a file.

We also used Observer, the protocol analyzer, to monitor the network as the emulator executed. Observer uses its own hardware resources to passively monitor the network so there was no difference in runtimes while executing the emulator. The runtimes were identical with, or without, Observer monitoring the network.

B. INFORMATION PROVIDED BY TECHNIQUES

We found that none of the techniques that we used, except the wrapping technique, can give us an estimate of the end-to-end latency and end-to-end throughput. Observer did provide time-based throughput, but only for those messages that used the network; any inter-process communication that did not use the network was not measured. Below we describe the types of output that we obtained from the various tools.

1. Pinger

The Pinger application sent small byte-size messages every thirty seconds while the emulator was executing. Four files were created on each machine. A log file was created that logged the start time and stop time of the program. Three other files were created that contained machine specific ping information. Each machine executed the Pinger application on itself, and the other two machines, while the emulator was running. However, the machine specific information that Pinger provided could not estimate the end-to-end status of communication resources. Pinger is only capable of alerting users and administrators of systems and workstations going down.

All three machines within the isolated domain executed the Pinger program every thirty seconds. Figure 4.2 contains an example of the information that was output to one of the files.

```
9/13/99 10:04:31 AM , 0 , 0 , 0 , 3 , 3
9/13/99 10:05:01 AM , 0 , 0 , 0 , 3 , 3
9/13/99 10:05:31 AM , 0 , 0 , 0 , 3 , 3
9/13/99 10:06:01 AM , 0 , 0 , 0 , 3 , 3
9/13/99 10:06:31 AM , 0 , 0 , 0 , 3 , 3
                    .
                    .
                    .
9/14/99 8:31:55 AM , 0 , 0 , 0 , 3 , 3
9/14/99 8:32:25 AM , 0 , 0 , 0 , 3 , 3
9/14/99 8:32:55 AM , 0 , 0 , 0 , 3 , 3
9/14/99 8:33:25 AM , 0 , 0 , 0 , 3 , 3
9/14/99 8:33:55 AM , 0 , 0 , 0 , 3 , 3
```

Figure 4.2. Pinger Machine Specific File Information

The machine specific information file contains the date, the average response time, the shortest response time, the longest response time, the number of packets sent, and the number of packets received. No other pertinent information was given or made available. The dates and times shown in Figure 4.2 are the actual dates and times the experiment took place. The program ran for approximately 24 hours and the ping responses were saved to this file. The actual size of the file created was approximately 105 KB. As previously stated, the communication overhead of Pinger was not substantial. The worst case was less than three percent.

2. WinDump

The WinDump application captured every packet that was transmitted through the network. The capture application received packets from the network and the packet information was written to a file. The file contained the source and destination nodes' IP address and port number, the TCP protocol flags, the data sequence number, the acknowledgement, the window of buffer space, and a maximum segment size option. Recall from Chapter III that our LAN configuration is used only by the traffic produced by the emulator. Regardless of that fact, WinDump produced a file that was over 1GB in size during the time it concurrently ran with the emulator. Additionally, the packet level information is very difficult to use to estimate the end-to-end status of communication resources. Another application would have to be developed to record, decipher, and

collate the collected information in real-time. Figure 4.3 contains an example of the packet level information that was written to the output file. Also, as previously stated, the overhead observed by the execution of the WinDump application was ten percent.

```
19:51:42.051959 ATROPOS.1036 > LUNA.139: P 667:706(39) ack 500 win 8261 (DF)
19:51:42.052556 LUNA.139 > ATROPOS.1036: P 500:539(39) ack 706 win 8055 (DF)
19:51:42.052699 ATROPOS.1036 > LUNA.139: P 706:749(43) ack 539 win 8222 (DF)
19:51:42.052896 LUNA.139 > ATROPOS.1036: P 539:582(43) ack 749 win 8012 (DF)
19:51:42.053030 ATROPOS.1036 > LUNA.139: F 749:749(0) ack 582 win 8179 (DF)
19:51:42.053118 LUNA.139 > ATROPOS.1036: F 582:582(0) ack 750 win  8012 (DF)
```

Figure 4.3. WinDump File Information

We now explain the WinDump data in Figure 4.3. As an example, the third line states, at 19:51:42.052699, TCP port 1036 on machine Atropos sent a packet to port 139 on machine Luna. The *P* means the *PUSH* flag is set in the packet. The packet sequence number is 706 and the packet contained 43 bytes of data. This packet also contains a piggy-backed acknowledgement from the packet on the previous line, acknowledging that packet's *PUSH*. The available receive window is 8222 bytes and the packet is marked with the trailing flag *Don't Fragment* (DF).

3. Observer

Observer also captured every packet that was transmitted through the network. However, Observer used its own hardware to monitor the network while the emulator executed. It passively captured packets from the network and measured the average throughput. It created an output file that was used to record the time and the minimum, average, and maximum throughput values. Figure 4.4 contains an example of the throughput information that was output to the file.

[Header]
Type=Bandwidth Utilization History
StartTime=937155407

[Data]

Time	Min	Avg	Max
11:04:17	0	21.4	32.9
11:04:47	7.1	20.3	32
11:05:17	0	4.4	14.2
11:05:47	0	18.3	32.6
11:06:17	7.8	23.2	31.1
11:06:47	0	4.6	13.7
11:07:17	0	15	32.5
11:07:47	8.5	24.4	31.5

Figure 4.4. Observer File Information

We now describe the Observer data. On line two, at 11:04:47, the minimum network throughput was 7.1 MB/s. The maximum throughput was 32 MB/s and the average throughput was 20.3 MB/s.

C. FUTURE WORK

In this thesis research, we modified an application emulator to record most of the information required to measure instantaneous throughput and latency as described in Schnaidt's thesis. In order to actually assess the accuracy of the wrapping measurements, a few `if` statements will need to be added to determine when (i) the early reader-late writer scenario has occurred and (ii) to record throughput only when the receipt of messages requires more than one invocation of the `receive` operation. Additionally, more fine grained information similar to that now sent to MSHN's RSS and RRD would need to be output. The testbed is already set up for this experiment with the appropriate version of the JVM installed.[2]

[2] One of the frustrations of this research work was that when using an earlier version of the JVM, JVM v. 1.2.1, an access violation would occur sporadically. This problem was resolved by running with JVM v. 1.3 Beta.

Additionally, several of the tools described in Chapter 2 were not used because they could not execute in a Windows NT environment. However, since the modified emulator was written in JAVA, if a purely UNIX testbed were available, the emulator could also be used with those tools to assess their accuracy.

D. SUMMARY

In this chapter, we presented our results from comparing various monitoring techniques. Three of the techniques that we used were passive and one was active. We measured the overhead of each and revealed that minimal overhead was added by most of the techniques examined. We also learned that only one of the techniques examined has the potential for predicting the end-to-end throughputs and latencies required by Carff's formulas [Ref. 18].

Finally, one of the major contributions of this thesis was to define a way to assess the accuracy of various communication performance monitoring tools, particularly in regards to their usefulness to RMSs, such as MSHN.

APPENDIX. SOURCE CODE

```
// Filename:    mc.java
// Version:     1.97
// Author:      Ron Jacobs
// Description:

//* masterControl.java
//*
//*
//*
//* main user interface for executing processes
//* USAGE: java masterControl <fileName>
//* The filename is optional, the default is a:\input.dec
//*
//* Read the data from given file and performs all possible schedules
//* each repeated statRepeat times.  This is used for statistical
//* computations.
//*
//* Each run is saved in a file scheduleX_Y.dat where X is the
//* schedule number and Y is the repeat time of that schedule.


import java.net.*;
import java.io.*;
import java.util.*;
import java.lang.Integer;
import java.awt.Frame;
import java.awt.TextArea;
import java.awt.event.*;


public class mc {
   public static void main(String [] args ) throws IOException {
      Frame statusFrame = new Frame("Master Control Status");
      statusFrame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent ev) {
            System.exit(0);
         }
      });
      TextArea statusText = new TextArea();
      statusFrame.add(statusText);
      statusFrame.setBounds(0,0,1000,400);
```

```
statusFrame.setVisible(true);


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// NUMBER OF TIMES TO REPEAT EACH SCHEDULE
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
int statRepeat = 30; //Number of times to repeat a schedule
                     //in order to get a valid set.


//Setup server socket to recieve data from machines
ServerSocket returnServerSocket = new ServerSocket(5000);
Socket returnSocket;

BufferedReader inFile;


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
// Get the name of the File to Open

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
String inputDecFilename;
if (args.length != 0){
   inputDecFilename = args[0];
}
else {
   inputDecFilename = "input.dec";
}
inFile = new BufferedReader(new FileReader(inputDecFilename));


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
// Setup variables needed when reading data in from file

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
int numberOfMachines;
int numberOfProcesses;
int numberOfSchedules = 1;
int[][] schedules;
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
    // For each run need the following
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
    StringBuffer scheduleDataFileName;  // Hold the name of the output File.
    int scheduleNumber = 1;


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
    // For each command need the following
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
    String myIpAddress =
      returnServerSocket.getInetAddress().getLocalHost().getHostName().toString();
    Vector commands = new Vector();
    StringBuffer commandString;
    String[] ip;
    String[] computeTime;
    String[][] data;
    int [][] inPort;
    int [][] outPort;
    int count;
    int[] counter;


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
    // Start reading data in from file
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
    numberOfMachines = Integer.parseInt(getFileLine(inFile)); //Used in schedules
    numberOfProcesses = Integer.parseInt(getFileLine(inFile));


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
    // Setup Schedule Table
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~

    for (int times = 0; times < numberOfProcesses; times++) {
      numberOfSchedules *= numberOfMachines;
    }
    schedules = new int[numberOfSchedules][numberOfProcesses];
    counter = new int[numberOfProcesses];

    for (int ix = 0; ix  < numberOfProcesses; ix++) {
      counter[ix] = 1;
    }


    for (int in = 0; in < numberOfSchedules; in++) {
      for (int p = 0; p < numberOfProcesses; p++) {
        schedules[in][p] = counter[p];
      }
      int carry = 0;
      for (int q = 0; q < numberOfProcesses; q++) {
        if (q == 0) {
          counter[q]++;
        }
        if (carry == 1) {
          counter[q]++;
          carry = 0;
        }
        if(counter[q] > numberOfMachines) {
          counter[q] = 1;
          carry = 1;
        }
      }
    }



//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
    // Instaniate Variables

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
```

```
        ip = new String[numberOfProcesses];
        computeTime = new String[numberOfProcesses];
        data = new String[numberOfProcesses][numberOfProcesses - 1];
        inPort = new int[numberOfProcesses][numberOfProcesses - 1];
        outPort = new int[numberOfProcesses][numberOfProcesses - 1];



//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
        // initialize inPort numbers begining at 6001

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~

        count = 6001;
        for(int ix = 0; ix < numberOfProcesses; ix++) {
           for (int iy = 0; iy < (numberOfProcesses - 1); iy++) {
              inPort[ix][iy] = count++;
           }
        }


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
        //initialize the outPorts

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~

        for(int ix = 0; ix < numberOfProcesses; ix++) {
           for (int iy = 0; iy < (numberOfProcesses - 1); iy++) {
              outPort[ix][iy] = ((iy+1)*numberOfProcesses - ix) + 6000;
           }
        }



//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
        // Get ip addresses of all machines

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
```

```
      for (int ix = 0; ix < numberOfMachines; ix++) {
         ip[ix] = getFileLine(inFile);
      }


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
      // Get process specific data

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
      for (int ix = 0; ix < numberOfProcesses; ix++) {
         computeTime[ix] = getFileLine(inFile);
         for(int iy = 0; iy < (numberOfProcesses - 1) ;iy ++) {
            data[ix][iy] = getFileLine(inFile);
         }
      }

      // Close the input file  Need to do this in order to copy to schedule
      // files.
      inFile.close();


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
      // Loop to run the schedules

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~

      //For testing swap the comment/uncomment of the following two lines and
      //and set the value in the second (currently 5) to the number of
      //schedules you want to run.
      for (int iv = 0; iv < numberOfSchedules; iv++) { //Do this for the number of
schedule
      //for (int iv = 1; iv < 3; iv++) {

         int[] tempSchedule = new int[numberOfProcesses];
         int[] transmitSchedule = new int[numberOfProcesses];
         int[] processArray;
         StringBuffer schedTxt = new StringBuffer();

         for (int ix = 0; ix < numberOfProcesses; ix++) {
            tempSchedule[ix] = schedules[iv][ix];
            transmitSchedule[ix] = schedules[iv][ix];
```

```
            schedTxt.append(tempSchedule[ix]);
            System.out.print(tempSchedule[ix]);
        }
        System.out.println("");


        //Create String Buffers for each command to be sent
        for (int pn = 0; pn < numberOfProcesses; pn++) {
            processArray = getPA(pn, numberOfProcesses);
            commandString = new StringBuffer();
            commandString.append(myIpAddress);
            commandString.append(" ");
            commandString.append(computeTime[pn]);
            commandString.append(" ");
            commandString.append(pn + 1);
            commandString.append(" ");

            for(int iy = 0; iy < numberOfProcesses - 1 ;iy ++) {
                commandString.append(inPort[pn][iy]);
                commandString.append(" ");
                commandString.append(outPort[pn][iy]);
                commandString.append(" ");
                commandString.append(
                    ip[tempSchedule[processArray[iy] - 1] - 1]);
                commandString.append(" ");
                commandString.append(data[pn][iy]);
                commandString.append(" ");
            }
            commands.addElement(commandString);

        }//end for create command


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~
        // For each schedule repeat statRepeat times for data that is
        // statistically valid.

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~

        for(int iz = 0; iz < statRepeat; iz++) {

            // Create a new fileName for storing data for this run
```

```
            // scheduleX_Y.dat
            scheduleDataFileName = new StringBuffer();
            scheduleDataFileName.append("schedule");
            scheduleDataFileName.append(scheduleNumber);
            scheduleDataFileName.append("_");
            scheduleDataFileName.append(iz + 1);
            scheduleDataFileName.append(".dat");


            Date ct = new Date(System.currentTimeMillis());
            statusText.append("\n" + ct.toString());
            statusText.append("\nOpening " + scheduleDataFileName.toString() + " for
output\n\n");

            System.out.print("\n" + ct.toString());
            System.out.println("\nOpening " + scheduleDataFileName.toString() + " for
output\n");

            BufferedWriter outFile = new BufferedWriter(
                        new FileWriter(scheduleDataFileName.toString()));

            //Write the header for the file (copy the input file)
            outFile.write("%" + scheduleDataFileName.toString());
            outFile.newLine();
            BufferedReader headerFile = new
              BufferedReader(new FileReader(inputDecFilename));

            try {
              while(true) {
                outFile.write("%");
                outFile.write(headerFile.readLine());
                outFile.newLine();
                outFile.flush();
              }
            }
            catch (NullPointerException e) {}

            headerFile.close();
            outFile.newLine();
            outFile.write("%%%%% Commands sent for this run %%%%%");
            outFile.newLine();
            outFile.newLine();
            outFile.write("%%" + ct.toString() + "%%");
            outFile.newLine();
            outFile.newLine();
```

```
        outFile.write("%% Closed " + scheduleDataFileName.toString() + " for
output%%");
        outFile.newLine();
        outFile.newLine();
        outFile.flush();

        long startTime = System.currentTimeMillis();
        for (int ix = 0; ix < numberOfProcesses; ix++) {

          Socket outSocket = new
            Socket(ip[transmitSchedule[ix] - 1], 5010);
          statusText.append("sending to " + ip[transmitSchedule[ix] - 1] + "\n");
          //System.err.println("sending to " + ip[transmitSchedule[ix] - 1]);
          BufferedWriter outCommand =
            new BufferedWriter(new
              OutputStreamWriter(outSocket.getOutputStream()));
          outCommand.write(commands.elementAt(ix).toString());
          outCommand.newLine();
          outCommand.flush();
          outSocket.close();
          outFile.write("%  ");
          outFile.write(commands.elementAt(ix).toString());
          outFile.newLine();
          outFile.flush();
        }

        //statusText.append("\n");
        outFile.write("%  ");
        outFile.write(schedTxt.toString());
        outFile.newLine();
        outFile.flush();

        //After issuing commands wait for the return values
        //Return Values include the time and the machine from which the
        //value came from.

        count = numberOfProcesses;
        outFile.newLine();
        outFile.write("%%%%% Data returned for this run %%%%%");
        outFile.newLine();
        outFile.newLine();
        outFile.flush();

        long maxTime = 0;
        while(count > 0) {
```

```
            returnSocket = returnServerSocket.accept();
            BufferedReader br =
              new BufferedReader(new
                InputStreamReader(returnSocket.getInputStream()));

            String timeAndMachine = br.readLine();
            statusText.append(timeAndMachine + "\n");
            //System.err.println(timeAndMachine);
            //write data to file
            outFile.write("%  ");
            outFile.write(timeAndMachine);
            outFile.newLine();
            outFile.flush();
            count--;
          }
          long stopTime = System.currentTimeMillis();
          outFile.newLine();
          outFile.write("%%%%% Time for this Schedule %%%%%");
          outFile.newLine();
          outFile.newLine();
          outFile.write(new Long(stopTime - startTime).toString());
          statusText.append("Total Run Time  = " +
            (new Long(stopTime - startTime).toString()) + "\n");
          outFile.newLine();
          outFile.flush();

          outFile.close();
          try {
            statusText.append("sleeping\n");
            Thread.sleep(5000);
            statusText.append("waking\n");
          }
          catch (InterruptedException e) {
            statusText.append("mc: " + e + "\n");
          }

        }//end repeat Times

        //close file

        commands.removeAllElements();
        scheduleNumber++;
      }//end For number of schedules
```

```
}//end main()

public static String getFileLine(BufferedReader inFile) throws IOException{
   String tempString = inFile.readLine();
   while((tempString.equals("")) || (tempString.charAt(0) == '%')) {
      tempString = inFile.readLine();
   }
   return tempString;
}

public static int[] getPA(int pn, int numP) {
   int[] tempArray = new int[numP-1];
   int counter = 0;
   for (int ix = 0; ix < numP; ix++){
         if (ix == pn){
         }
         else {
               if (counter < numP) {
                     tempArray[counter] = ix + 1;
                     counter++;
               }
               else {
                  //statusText.append("COUNTER TOO BIG");
                  System.err.println("COUNTER TOO BIG");
               }
         }
   }
   return tempArray;
}



public static int[] fwdIterate(int[] temp) {
   int leng = temp.length;
   System.err.println(leng);
   int tempStore = temp[0];
   for (int ix = 0; ix < (leng-1); ix++) {
      temp[ix] = temp[ix + 1];
   }
   temp[leng - 1] = tempStore;
   for (int g = 0; g < leng; g++){
      System.err.print(temp[g]);
   }
```

```
        System.err.println("");
        return temp;

    }

    public static int[] revIterate(int[] temp) {
        int leng = temp.length;
        System.err.println(leng);
        int tempStore = temp[leng-1];
        for (int ix = (leng-1); ix > 0; ix--) {
            temp[ix] = temp[ix-1];
        }
        temp[0] = tempStore;
        for (int g = 0; g < leng; g++){
            System.err.print(temp[g]);
        }
        System.err.println("");
        return temp;

    }

}//end class
```

```
// Filename:        appem.java
// Version:             1.96
// Author:          Ron Jacobs
```

```
// Description:
//
// Program simulates a process.  It emulates the behavior of a
// process based on number of messages, message
// frequency distribution, message size and message size
// distribution.
//
// For n processes, a given process may communicate with n-1
// other process.  An inmsg, and outmsg thread is started.
//


package appem;

import java.net.*;
import java.io.*;

public class appem {

   public static long outmsgTime;
   public static long outlatencyTime;
   public static long inmsgTime;
   public static long inlatencyTime;
   public static long determinemsgSize;

   public static void main(String [] args) throws IOException {

      long timeAppem1 = System.currentTimeMillis();
      long timeAppem2;
      long timeAppem3;
      long timeAppem4;
      long timeAppem5;
      long timeAppem6;

      long runTimeAppem1;
      long runTimeAppem2;
      long runTimeAppem3;
      long runTimeAppem4;
      long runTimeAppem5;
      long runTimeAppem6;

      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      //setup variables
      //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      int numOtherProcesses = (args.length - 3)/12;
```

```
int[]inPort     = new int[numOtherProcesses];
int[] outPort   = new int[numOtherProcesses];
String[] ipAdd  = new String[numOtherProcesses];
int computeTime;
int[] numMessages = new int[numOtherProcesses];
int[] numMsgDist  = new int[numOtherProcesses];
int[] numMsgParam1 = new int[numOtherProcesses];
int[] numMsgParam2 = new int[numOtherProcesses];
int[] msgSize     = new int[numOtherProcesses];
int[] msgSizeDist = new int[numOtherProcesses];
int[] msgSizeParam1 = new int[numOtherProcesses];
int[] msgSizeParam2 = new int[numOtherProcesses];
int[] sync = new int[numOtherProcesses];

Thread[] inmsgThreads = new Thread[numOtherProcesses];
Thread[] outmsgThreads = new Thread[numOtherProcesses];


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//get data from command line
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
computeTime = Integer.parseInt(args[1]);
String pid = args[2];
int pidInt = new Integer(pid).intValue();

int Index = 3;
for(int ix = 0; ix < numOtherProcesses; ix++) {
   inPort[ix] = Integer.parseInt(args[Index]);
   outPort[ix] = Integer.parseInt(args[Index + 1]);
   ipAdd[ix] = args[Index + 2];
   numMessages[ix] = Integer.parseInt(args[Index + 3]);
   numMsgDist[ix] = Integer.parseInt(args[Index + 4]);
   numMsgParam1[ix] = Integer.parseInt(args[Index + 5]);
   numMsgParam2[ix] = Integer.parseInt(args[Index + 6]);
   msgSize[ix] = Integer.parseInt(args[Index + 7]);
   msgSizeDist[ix] = Integer.parseInt(args[Index + 8]);
   msgSizeParam1[ix] = Integer.parseInt(args[Index + 9]);
   msgSizeParam2[ix] = Integer.parseInt(args[Index + 10]);
   sync[ix] = Integer.parseInt(args[Index + 11]);
   Index += 12;
}


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
// Initialize buffer - read in from file.
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

String buf = "";
try{
   BufferedReader is =
      new BufferedReader(new FileReader("msgfill"));
   buf = is.readLine();
}
catch (FileNotFoundException e) {
}

timeAppem2 = System.currentTimeMillis();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//Start the inmsg Threads.
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
for(int ix = 0; ix < numOtherProcesses; ix++) {
   inmsgThreads[ix] = new inmsg(inPort[ix], pidInt, ix);

}


timeAppem3 = System.currentTimeMillis();

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Start outmsg Threads
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
for(int ix = 0; ix < numOtherProcesses; ix++) {
   outmsgThreads[ix] =
      new outmsg(ipAdd[ix], outPort[ix], msgSize[ix],
         msgSizeDist[ix], msgSizeParam1[ix], msgSizeParam2[ix],
         numMessages[ix], buf, pidInt, ix);

}

timeAppem4 = System.currentTimeMillis();


//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Join all Threads
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

try {
```

```
    for(int ix = 0; ix < numOtherProcesses; ix++) {
      outmsgThreads[ix].join();
      //stat.append("outmsg" + ix + " complete\n");
    }
  }

  catch (Exception e){
    //stat.append("error appem outmsg:  " + e);
    while(true) {}
  }

  timeAppem5 = System.currentTimeMillis();

  try{
    for(int ix = 0; ix < numOtherProcesses; ix++) {
      inmsgThreads[ix].join();
      //stat.append("inmsg" + ix + " complete\n");
    }
  }

  catch (Exception e){
    //stat.append("error appem inmsg:  " + e);
    while(true) {}
  }

 timeAppem6 = System.currentTimeMillis();


 //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 //calculate runtimes
 //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 runTimeAppem1 = timeAppem2 - timeAppem1;
 runTimeAppem2 = timeAppem3 - timeAppem2;
 runTimeAppem3 = timeAppem4 - timeAppem3;
 runTimeAppem4 = timeAppem5 - timeAppem4;
 runTimeAppem5 = timeAppem6 - timeAppem3;
 runTimeAppem6 = timeAppem6 - timeAppem1;


 //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 //Section returns the necessary data
 //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

 String returnIP = args[0];
```

```
        Socket sock = new Socket(returnIP, 5000);
        BufferedWriter sendValue =
          new BufferedWriter(
            new OutputStreamWriter(sock.getOutputStream()));
        StringBuffer returnString = new StringBuffer();
        returnString.append("In Msg Latency Time " + new
Long(inlatencyTime).toString());
        returnString.append("  ");
        returnString.append("In Msg Time " + new Long(inmsgTime).toString());
        returnString.append("  ");
        returnString.append("Out Msg Latency Time " + new
Long(outlatencyTime).toString());
        returnString.append("  ");
        returnString.append("Out Msg Time " + new Long(outmsgTime).toString());
        returnString.append("  ");
        returnString.append("Determine Msg Size " + new
Long(determinemsgSize).toString());
        returnString.append("  ");
        returnString.append("Start Main Latency Time " + new
Long(runTimeAppem1).toString());
        returnString.append("  ");
        returnString.append("Start Main Inmsg Thread " + new
Long(runTimeAppem2).toString());
        returnString.append("  ");
        returnString.append("Start Main Outmsg Thread " + new
Long(runTimeAppem3).toString());
        returnString.append("  ");
        returnString.append("Inmsg Joined " + new Long(runTimeAppem4).toString());
        returnString.append("  ");
        returnString.append("Outmsg Joined " + new Long(runTimeAppem5).toString());
        returnString.append("  ");
        returnString.append("Total Runtime " + new Long(runTimeAppem6).toString());
        returnString.append("  ");
        returnString.append(
          sock.getInetAddress().getLocalHost().getHostName().
            toString());
        returnString.append("  ");
        returnString.append(pid);
        sendValue.write(returnString.toString());
        sendValue.newLine();
        sendValue.flush();
        sendValue.close();
        System.exit(0);
    }
  }
```

```
// Filename:   distibution.java
// Version:    1.96
// Author:     Ron Jacobs
// Description:
//   This will return an random integer number based
//   on a mean value (mean) and distribution (d).

package appem;

import java.util.*;

class distribution {

   public distribution() {
   }


   public int value(int mean, int d, int stdDev, int p2) {
      int returnValue;
      Random rn = new Random(System.currentTimeMillis());
      switch(d) {
         case 1: //constant
            returnValue  = mean;
            break;
         case 2: //exponential
            double u = rn.nextDouble();
            returnValue = (int)(0-(mean * Math.log(u)));
            break;
         case 3:
            returnValue = mean;
            break;
         default:
            returnValue = mean;
            break;
      }
      return returnValue;
   }

}
```

```
// Filename:    inmsg.java
// Version:     1.96
// Author:      Ron Jacobs
// Description:
//   Message receiver for thesis program.  It receives strings from
//   the respective outMsg. It checks the message for the termination
//   sentinal (999).  If it is the termination sentinal it sets the a
//   boolean to false to terminate the looping which terminates the
//   thread.
//


package appem;

import java.io.*;
import java.net.*;


class inmsg extends Thread  {


   long timeInmsg1 = System.currentTimeMillis();
   long timeInmsg2;
   long timeInmsg3;

   int inPort;
   int pid1;
   int pid2;
   boolean moreToReceive;

   public inmsg(int iPt, int pid1, int pid2) {

      this.pid1 = pid1;
      this.pid2 = pid2;
      this.inPort = iPt;
      moreToReceive = true;
      start();


   }

   public void run() {
```

```
try {

   ServerSocket listen_socket = new ServerSocket(inPort);
   Socket sock;

   String buf;
   sock = listen_socket.accept();
   BufferedReader is =
      new BufferedReader(new InputStreamReader(sock.getInputStream()));

   timeInmsg2 = System.currentTimeMillis();

         while (moreToReceive) {
      timeInmsg3 = System.currentTimeMillis();
      //read data in from socket
      buf = is.readLine();

      if (buf.equals("111")) {

         timeInmsg3 = timeInmsg3 +
            (System.currentTimeMillis() - timeInmsg3);
          moreToReceive = true;
      }
      else if

       //check for termination sentinal
            (buf.equals("999")) {

         timeInmsg3 = timeInmsg3 +
            (System.currentTimeMillis() - timeInmsg3);
          moreToReceive = false;
      }
      else
      {
        //continue to read
           timeInmsg3 = timeInmsg3 +
           (System.currentTimeMillis() - timeInmsg3);
          moreToReceive = true;
      }


   }


  sock.close();
```

```
            appem.inlatencyTime = timeInmsg2 - timeInmsg1;
            appem.inmsgTime = timeInmsg3 - timeInmsg2;



        }
        catch (Exception e) {
            //moreToReceive = true;
            //while(true){}

        }
    }
}
```

```
// Filename:   outmsg.java
// Version:    1.96
// Author:     Ron Jacobs
// Description:
//   This program sents messages to its respective inMsg thread.
//   Using the shared waitDevice, shared with calc, it sends messages
//   for each increment of the msg variable in waitDevice and
//   continues until msg is zero and boolean keepgoing (in waitDevice
//   also) is false.  It then sends one last message which contains
//   the termination sentinal.
//


package appem;

import java.io.*;
import java.net.*;

class outmsg extends Thread {


   long time1 = System.currentTimeMillis();
   long time2;
   long time3;
   long time4;

   String ip;
   int outPort;
   int msgSize;
   int msgSizeDist;
   int param1;
   int param2;
   int nextMsg;
   int nummsg;
   String buf;

   distribution distGen;


   public outmsg(String ip, int op, int ms, int msd, int p1, int p2,

      int nm, String buf, int pid1, int pid2) {


      this.ip = ip;
```

```
    this.outPort = op;
    this.msgSize = ms;
    this.msgSizeDist = msd;
    this.param1 = p1;
    this.param2 = p2;
    this.nummsg = nm;
    this.buf = buf;
    distGen = new distribution();
    start();



}


public void run() {

    boolean socketClosed = true;
    boolean moreWork = true;

    while (socketClosed) {

      try {
        Socket sock = new Socket(ip, outPort);
        BufferedWriter out =
          new BufferedWriter(new
            OutputStreamWriter(sock.getOutputStream()));
        socketClosed = false;
        time2 = System.currentTimeMillis();
        time4 = System.currentTimeMillis();

        for (int ix = 0; ix < nummsg; ix++) {
                //determine size of the next message
          nextMsg = distGen.value(msgSize, msgSizeDist, param1, param2);
           time4 = time4 + (System.currentTimeMillis() - time4);

            time3 = System.currentTimeMillis();

             out.write("111");
             time3 = time3 + (System.currentTimeMillis() - time3);
             out.newLine();
             out.write(buf, 0, nextMsg);
             out.newLine();
```

```
            }

            out.write("999");
            time3 = time3 + (System.currentTimeMillis() - time3);
            out.newLine();
            out.flush();
            out.close();
            socketClosed = false;

            appem.outlatencyTime = time2 - time1;
            appem.outmsgTime = time3 - time2;
            appem.determinemsgSize = time4 - time2;


          }
          catch (Exception e) {
            socketClosed = true;
            while (true) {}
          }
        }
      }
    }
```

```
// Filename:    server.java
// Version:     1.96
// Author:      Ron Jacobs
// Description:
//    server for each machine.  This is to be started on each machine
//    that will be receiving commands to start mainProcesses.  It
//    simply receives a string that is concatinated with another
//    string to be exec'd and start the mainProcess.
//

package server;

import java.net.*;
import java.io.*;

public class server {
   public static void main(String [] args ) throws IOException {

      StringBuffer commandString;
      ServerSocket ss = new ServerSocket(5010);

      while(true){
         commandString = new StringBuffer();
         commandString.append("java appem.appem ");
         Socket sock = ss.accept();
         BufferedReader br =
            new BufferedReader(
               new InputStreamReader(sock.getInputStream()));
         commandString.append(br.readLine());
         System.out.println("Sending: " +
            commandString.toString());
         Process p =
            Runtime.getRuntime().exec(commandString.toString());
      }
   }
}
```

LIST OF REFERENCES

[1] W. Stallings. *High-speed Networks: TCP/IP and ATM Design Principles.* Prentice-Hall Inc., Upper Saddle River, NJ, 1998.

[2] M. C. L. Schnaidt. Design, implementation, and testing of MSHN's application resource monitoring library. Master's thesis, Naval Postgraduate School, Monterey, CA, December 1998.

[3] J. Kresho. Quality Network load information improves performance of adaptive applications. Master's thesis, Naval Postgraduate School, Monterey, CA, December 1998.

[4] B. Supinski, N. Karonis. *Accurately measuring MPI broadcasts in a computational grid. Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 29-37, August 1999.

[5] N. Kapadia, J. Fortes, C. Brodley. *Predictive Application-Performance Modeling in a Computation Grid Environment. Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 47-54, August 1999.

[6] P. Dinda, D. O'Hallaron. *An Evaluation of Linear Models for Host Load Prediction. Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 87-96, August 1999.

[7] R. Wolski, N. Spring, and J. Hayes. *Predicting the CPU Availability of Time-shared Systems on the Computational grid. Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 105-112, August 1999.

[8] B. Lowekamp, D. O'Hallaron, T Gross. *Direct Queries for Discovering Network Resource Properties in a Distributed Environment. Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 38-46, August 1999.

[9] H. Fang, C. Hagwood. Dynamic adaptive modeling for the performance of the end-to-end network measurements and estimation. Technical Report, NIST, March 1999.

[10] Netperf. Netperf Revision 2.1. Manual, 1996. Available at www.netperf.com

[11] Gloperf. Installing the Globus Metacomputing Toolkit. Manual, 1999. Available at www.globus.com

[12] SNMP. SNMP for Dummies. Manual, 1999. Available at www.baynetworks.com

[13] T. Gross, P. Steenkiste, J. Subhlok. *Adaptive Distributed Applications on Heterogeneous Networks. Proceedings of the Eighth Heterogeneous Computing Workshop*, pages 209-217, August 1999.

[14] R. Wolski, N. Spring, C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. Sc97 Technical paper, University of California, San Diego, Computer Science and Engineering Department, 1997.

[15] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. Technical report, Lawrence Berkeley National Laboratory, University of California, Berkeley, Computing Sciences Directorate, 1998.

[16] WinDump. Installing the WinDump NDIS packet capture driver. Manual, 1999. Available at www.netgroup-serv.polito.it

[17] Observer. LAN analyzer and troubleshooting tool. Manual, 1998. Available at www.networkinstruments.com

[18] P. Carff. When is a simple model adequate for use in scheduling in MSHN. Master's thesis, Naval Postgraduate School, Monterey, CA, March 1999.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Chairman, Code CS.....1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

4. Dr. Debra Hensgen.....2
   Computer Science Department, Code CS/HD
   Naval Postgraduate School
   Monterey, California 93943-5100

5. Dr. Taylor Kidd.....2
   Computer Science Department, Code CS/HD
   Naval Postgraduate School
   Monterey, California 93943-5100

6. Ron Jacobs.....5
   6999 North Loop Drive, Res #3
   El Paso, Texas 79915