1999

Passive, Domain-Independent, End-to-End Message Passing Performance Monitoring to Support Adaptive Applications in MSHN

Schnaidt, Matt; Hensgen, Debra; Falby, John; Kidd, Taylor; St. John, David

Passive, Domain-Independent, End-to-End Message Passing Performance Monitoring to Support Adaptive Applications in MSHN[r]

Matt Schnaidt
Debra Hensgen
John Falby
Taylor Kidd
David St. John

Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5118

Abstract

This paper focuses on the problem of monitoring the end-to-end performance of message passing to support adaptive applications to be executed using the MSHN system (Management System for Heterogeneous Networks). Eight commercial and research tools and application components that attempt to measure perceived end-to-end message passing performance were identified. Two were dismissed; one because of recently published findings and the other because it is typically used in too many inconsistent configurations. The remaining six are carefully described in the paper. We were able to characterize each as either passive or active, determine whether they require domain-specific knowledge of an application, identify sources of inaccuracies, and enumerate their limitations. Based upon this survey, and previous analytical experiments, we conclude that the optimal monitoring mechanism: (1) should be passive; (2) should not require domain-specific knowledge of an application; (3) should minimize sources of error; and (4) should have few limitations. No single tool or application component surveyed has all of these characteristics. Based upon the surveyed work and other recent research in distributed systems, we have synthesized a new tool whose mechanisms have all of the desired characteristics. This paper describes our mechanism, and how we implemented it, in detail.

1. Introduction

Any system managing a set of distributed heterogeneous resources, whether a native distributed operating system or a resource management system, must maintain status information concerning those resources. Some of the status information is slowly changing, such as the speed of a particular CPU. Some of the information may change more quickly, such as the version of the operating system, the type of security services,

[r] This research is supported by DARPA Order E-583 under the DARPA QUORUM program and also, in part, by the Institute for Joint Warfare Analysis

and the type of network card. Finally, some status information will change very quickly such as the current length of the CPU queue, the amount of available memory, and the current load on a network. Though possibly cumbersome, the first two types of information can be manually entered into the management system's database when new hardware or software is installed. The third type of information, however, changes so quickly that there must be a method for automatically collecting it. The Management System for Heterogeneous Networks (MSHN[1]), an experimental distributed resource management system (RMS) that we are building, requires an estimate of this quickly changing information. Because users can place loads on some of the resources within MSHN's venue without submitting requests to MSHN, MSHN requires a mechanism that can accurately estimate the *end-to-end* availability of each of the resources. This paper documents our efforts to locate, and eventually, synthesize a mechanism to provide this information for the network resources. In this section, we first briefly describe the MSHN project, its place within the larger QUORUM program, and the reason why MSHN requires end-to-end status information. We then describe exactly what status information we currently need as well as the constraints under which a mechanism to obtain this information must operate.

1.1 MSHN

In order to put the research described in this paper into perspective we provide a brief overview of MSHN and we summarize the QUORUM program, under which MSHN is a project. The goal of the QUORUM program is to develop a software architecture, consisting of translucent layers, that delivers good end-to-end quality of

[1] Pronounced "mission"

service (QoS) to a dynamically changing set of adaptive applications that are competing for resources within a distributed, heterogeneous computing infrastructure. QUORUM consists of many major research projects including ones that: (1) define languages and models for expressing user-level QoS; (2) design and construct tools that convert user-level QoS to resource requirements; (3) design and construct languages and databases for describing resource requirements and resource characteristics; (4) design and construct RMS's; (5) define mechanisms for achieving translucence; (6) design appropriate feedback mechanisms; (7) define benchmarks to be used as representatives of future adaptive applications; and (8) research new ideas in distributed operating systems and network protocols. MSHN is one of several RMSs being designed, implemented, and tested under QUORUM. MSHN focuses on four basic areas: (i) the granularity of resources and the protocols and policies used to allocate them required by RMS's to derive good schedules for adaptive applications; (ii) heterogeneous scheduling algorithms; (iii) estimating the available resources as well as the required resources from historical and recently collected resource usage data; and (iv) determining how to ensure that RMSs remain in a stable state.

Given a set of jobs, MSHN will determine where and when to run each job along with the appropriate version of the job to run. MSHN evolved from SmartNet, which was a heterogeneous framework for minimizing the time at which the last job of a set of computationally intensive jobs finishes on a suite of heterogeneous computing resources [KIDD96]. SmartNet treated the set of compute resources available as one virtual heterogeneous machine (VHM). SmartNet achieved superior performance by mapping applications to resources based upon knowledge of the VHM and job characteristics.

MSHN differs from SmartNet in several ways: (1) it strives to support Input/Output intensive and real-time jobs, in addition to compute-intensive jobs; (2) it accounts for the fact that a job may need many different resources, not just a CPU, to execute; and (3) it manages adaptive applications.

One of the important improvements of MSHN over traditional RMS's is that MSHN will support adaptive applications. Adaptive applications are those that can produce results using one of a variety of algorithms or in one of a variety of forms.

MSHN has a client-server architecture. It is composed of a Client Library, a Scheduling Advisor, a Resource Requirements Database, a Resource Status Server, and a MSHN Daemon.

The following paragraphs provide an abstract description of each of the components, and Figure 1 provides an overview of the entire architecture. Although these components are shown together, they may in fact reside on separate machines, and, in certain situations, be replicated. Usually, many different client applications will be running at any given time.

The Client Library

The client library is linked with both adaptive and non-adaptive applications. It provides a transparent interface to all of the MSHN services [KRES97]. The client library performs at least the following functions: (1) it intercepts system calls to record resource requirements; (2) it forwards requests to start another process, when appropriate, to the Scheduling Advisor; and (3) it intercepts and performs the appropriate action on requests from the Scheduling Advisor to adapt. It forwards the recorded resource requirements to the Resource Requirements Database. The final implementation

of MSHN will be able to forward the performance measurements and resource requirements through the MSHN daemon when that is more efficient.

The Scheduling Advisor

The Scheduling Advisor performs the highly complex task of scheduling multiple jobs, from multiple users, onto one (or several) computers from a pool of heterogeneous computing platforms. The sophisticated algorithms that the Scheduling Advisor will use to make decisions are beyond the scope of this paper. However, this research requires knowledge of the interfaces presented by the Scheduling Advisor.

The Scheduling Advisor will accept scheduling requests from the client libraries. The Scheduling Advisor will query both the Resource Status Server and the Resource Requirements Database. These queries must respond with near real-time information on the status (load) of the distributed resources, and the resource requirements of the application. Once the Scheduling Advisor receives this load information, it can calculate, if possible, a mixture of computing and network resources that will, with high probability, deliver the requested quality of service.

Additionally, in the event of a significant deviation from the initial resource status estimate, the Scheduling Advisor will receive notification from the Resource Status Server. For example, if a communications path is severed, or a machine fails, the Scheduling Advisor will be notified and can recalculate a new scheduling solution for the affected applications. The Scheduling Advisor may then signal the client library and advise it that the application should begin using a different algorithm, or perhaps recommend that it shift execution to a different set of resources. The granularity of

resource model needed by the Scheduling Advisor is a topic of major research in the MSHN project.

The Resource Requirements Database

The Resource Requirements Database is a repository of information pertaining to the execution of user applications. This database contains statistics on the run time characteristics of jobs, such as CPU, memory, and disk usage. The Resource Requirements Database provides this information to the Scheduling Advisor upon request. While currently the MSHN client library is its only source of information, we envision that other tools, currently under development within the QUORUM project could also provide information for this database.

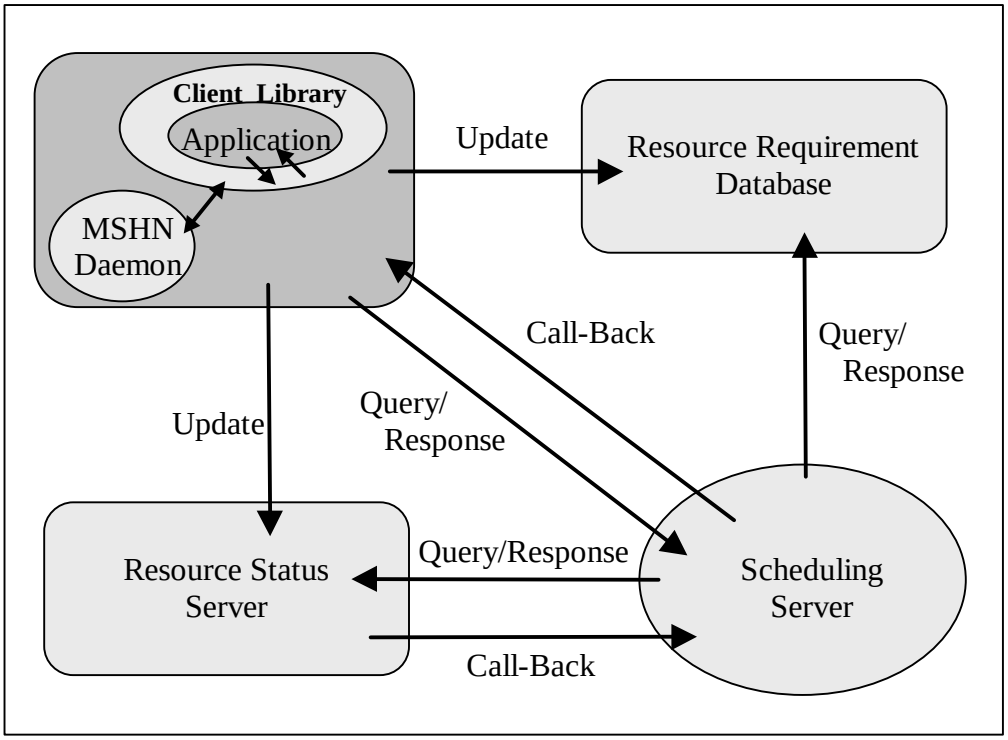

Figure 1: MSHN Architecture

The Resource Status Server

The purpose of the Resource Status Server is to maintain a repository of the three types of information about the resources available for MSHN to schedule: the relatively static, the moderately dynamic, and the highly dynamic information. The Scheduling Advisor will query the Resource Status Server to obtain an initial estimate of the currently available computing and networking resources. After making a scheduling decision, the Scheduling Advisor will notify the Resource Status Server of the additional loads that it expects the client application to place on the compute and networking resources. Much of this paper is dedicated to determining the best mechanisms for obtaining this most dynamically changing type of information for network resources.

The MSHN Daemon

The MSHN Daemon executes on all compute resources available for scheduling by the MSHN Scheduling Advisor. It is used to begin and control the execution of processes that are submitted to MSHN. It is also used to filter information from the client libraries which is destined for the Resource Requirements Database or Resource Status Server when several different MSHN jobs are executing locally.

1.2 Constraints

MSHN requires the gathering of resource usage information for applications that run within the MSHN system as well as status information for resources within the scope of the MSHN Scheduling Advisor. The MSHN Scheduling Advisor uses this information to make scheduling decisions. The methods used to gather this information are subject to three constraints: (1) the implementation must not require any changes to an operating

system; (2) modifications to the application code must be minimized; and (3) the overhead imposed by the information gathering mechanism should not be excessive.

There are many reasons for the first requirement. Our ultimate goal is the widespread acceptance of MSHN-type systems. Many potential users are reluctant to use systems or tools that require that their operating system be modified. 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. Addtionally, we do not want to risk compromising the security features of the operating system by changing the kernel. Finally, source code of all operating system releases may not be available.

The second and third requirements address acceptance and usability issues. If the application writer must modify his code, or if the use of our system incurs unacceptable overhead, the system will not be used.

1.3 Organization of Remainder of Paper

The next section of this paper surveys existing mechanisms that we considered as candidates for MSHN to use to estimate end-to-end network status. That section describes the six that we considered as initially viable candidates in some detail and summarizes their similarities and differences. In section 3, we describe the mechanism that we are currently using in MSHN, which synthesizes the best attributes that we found in our survey, as well as other recent theoretical and practical distributed system results. Finally, we summarize our findings – including giving an empirically obtained measurement of the worst-case overhead incurred and how we can reduce it – and outline

the set of experiments that we are currently conducting to assess the accuracy with which we can predict quickly changing resource availability.

2. Existing Mechanisms for Estimating Network Availability

To support adaptive applications, the MSHN Scheduling Advisor requires end-to-end status information for the resources at its disposal. One critical resource is the network. The remainder of this paper addresses the issue of monitoring the current network availability. We first review several tools, protocols, and application components that estimate network availability and then summarize the desirable characteristics of these systems.

To support MSHN's end-to-end network monitoring, we initially considered eight application components, protocols, and tools, from both the commercial and research sectors: `Ping`, `ftp` (the File Transfer Protocol application), Netscape Communicator, Network Weather Service (NWS), Netperf, BBN's Communications Server (Commserver), Resource Reservation Protocol (RSVP) and Simple Network Management Protocol (SNMP). We quickly rejected two as inappropriate for MSHN: SNMP and RSVP. We could not directly use the Simple Network Management Protocol (SNMP) in MSHN because it provides link-based information and does not estimate end-to-end throughput and latency between machines on remote, nonadjacent networks [PERK97]. Any tool built on top of SNMP would necessarily have to change when routing algorithms changed. We also considered RSVP, but have rejected it largely due to recent results indicating that the bandwidth it allocates can be substantially different from that requested[LEEC98]. We now discuss each of the six remaining in detail and then compare and contrast advantages and disadvantages of each type.

2.1 Ping

The `ping` program is ubiquitous. Its primary use is for troubleshooting networks. In its default configuration, if a network connection exists between two machines, the execution of `ping` results in a short message making a round trip from the local to the remote host and back. When `ping` completes, it prints the number of bytes sent and the round trip time. Thus, `ping` provides a single packet measurement of network throughput.

`Ping` packages its local timestamp in an Internet Control Message Protocol (ICMP) packet and sends it to the targeted host. The targeted host receives this datagram, its IP layer recognizes the type (ECHO_REQUEST) and immediately repackages the data contained in the packet, sending it back to the pinger. The pinger receives the reply datagram and subtracts its machine's current time from the timestamp that the pinger placed in the original datagram, thus determining round trip time. [BERK91]

2.2 The File Transfer Protocol Application

The File Transfer Protocol application (`ftp`) is used to transfer files between machines connected by a network. We studied `ftp` because it estimates end-to-end (meaning application to application) throughput after completing a file transfer. The `ftp` application is a client-server application; the `ftp` server runs as a background process listening on a fixed port for client connection requests. The user starts the `ftp` client in order to issue requests to the `ftp` server on the remote machine. Because of the tight coupling between the `ftp` server and client, it is possible for `ftp` to estimate the throughput associated with transferring a file. We now summarize the actions that occur

in transferring a file, F, from the `ftp` server on computer A to the `ftp` client on computer B.

The `ftp` server on computer A listens on port Y. The `ftp` client on computer B connects to the server on computer A at port Y. The server on computer A accepts the connection and generates a child process that will handle all future interactions with the `ftp` client via the connection. We will call this connection the control connection. The `ftp` client on computer B sends a request across the control connection to the `ftp` server on computer A to send file F. Included in the request to A is the port number, X, that the client on computer B will listen to. The `ftp` client on B then listens on port X. The server sends the size, S, of file F to the `ftp` client over the control connection. The client receives S. The server connects to the client on computer B at port X. The client accepts the connection and records computer B's time as $T_{startRead}$. We will call this connection the data connection. Once the server sees that the client accepted the data connection, it sends the file, F, across that connection and the client reads until it receives the entire file. When the entire file has been received, the client records computer B's time as $T_{endRead}$. The client then uses S and the time elapsed between $T_{startRead}$ and $T_{endRead}$ to estimate throughput.

2.3 Netscape Communicator

Netscape recently made the source code for their web browser freely available [NETS98]. By examining this source code, we learned that they use an approach similar to `ftp`'s to calculate throughput. Netscape Navigator displays a "thermometer" at the bottom of the browser, showing the user the current download speed, the amount of the file already downloaded, and the estimated time to complete the download. When

downloading large files, the system call `read()`must typically be invoked more than once by the browser. Following each invocation of `read()`, Netscape updates the thermometer's data fields. The throughput estimate displayed on the thermometer is cumulative in that each calculation is based upon the total number of bytes downloaded, the total size of the file to download, and the total time since the first `read()` was called for the current file.

2.4 Network Weather Service

The Network Weather Service (NWS) is a tool for predicting computer and network performance for use by metacomputing applications [WOLS97] [SPRI97]. NWS makes periodic estimates of availability of resources for which it is responsible. One of the estimates made by NWS is network availability, specifically, the latency and throughput observed between two computers.

In order to measure the latency between two computers, A and B, a NWS process on computer A sends a small message to a corresponding process on computer B. The process on B immediately replies to the process on A, with the process on A recording the round trip time. NWS approximates latency as half of this round trip time.

To estimate throughput, the NWS process on host A sends a large message to the corresponding process on host B, and the process on B sends a small acknowledgement message back to the process on A. The NWS process on computer A estimates the transmission time of the large message as the round trip time, less the latency estimate described above. Throughput is then estimated as the message size divided by estimated transmission time. NWS keeps a record of previous estimates of throughput and latency, which it uses to predict future resource availability using statistical modeling techniques.

The developers use a token passing scheme to avoid overloading the network. They note that token passing does not scale well with large distributed systems. Therefore, the accuracy of the throughput and latency estimates degrade as the size of the network increases. Additionally, token passing can introduce security problems [STAL98]. Finally, to capture fluctuations in network QoS, the developers note that administrators must increase both message size and monitoring frequency.

2.5 Netperf

Netperf is a benchmark that can be used to measure different aspects of network availability with a primary focus on actively measuring the throughput of bulk data transfers and the request/response round trip time [HEWL96]. Netperf contains a rich benchmarking suite with many options for simulating specific scenarios (e.g., http protocols). Netperf's bulk data transfer test can be used to estimate the throughput between a local and remote host communicating over a network. It works by sending data for a period of time (the default is 10 seconds), and then measuring the total amount of data sent and received after the time period has elapsed. Netperf's request/response time test is very similar to that used by NWS to estimate latency: a short message is sent from a local to a remote host; the remote host replies immediately; and the local host measures round trip time and approximates latency as one half of this round trip time.

2.6 BBN's Commserver

The Joint Task Force Advanced Technology Demonstration (JTF ATD) strives to predict trends in the advances of future hardware and software. It also provides a reference architecture into which such advances can be easily integrated. At the base of the JTF architecture is the Commserver whose ultimate purpose is to permit applications

to be network aware. The job of the Commserver is to estimate the available bandwidth between JTF users. The Commserver uses that bandwidth, along with the priorities of various users (expressed as currency), and a list of the applications requiring execution to allocate resources. [HAYE94]

Early experimentation with the JTF ATD Commserver [KRES98] revealed several problems. First, the Commserver places a load on the network in order to estimate end-to-end latency and bandwidth available. Second, it uses the throughput and latency estimates directly, without reference to previous measurements, to estimate network load. Due to the rapidly changing nature of network traffic, this technique can return inaccurate or misleading results. Finally, the estimates returned are inaccurate unless the network is sampled frequently which further increases the network load.

2.7 Characteristics of These Systems

We divide the mechanisms described above into two categories: **passive** and **active**. Active mechanisms place additional loads on the resources that they are monitoring; passive ones do not. Applying this definition, we see that `ping`, NWS, Netperf and BBN's Commserver all use active mechanisms, while `ftp` and Netscape use passive mechanisms. Unfortunately, it is when the network is most busy that we need the most accurate estimates. This is when there is no extra bandwidth available to give to these active mechanisms. Passive mechanisms, on the other hand, do not add to the load carried by the already scarce resource. For this reason, in MSHN we prefer passive monitoring.

Another way of categorizing the previously discussed tools and application components for measuring network performance attributes is to consider how closely tied

they are to applications. We note that the programmers of both `ftp` and Netscape used domain-specific knowledge to obtain estimates of network throughput. MSHN prefers that application writers not be required to also worry about measuring resource availability; availability should be measured by the system.

Finally, there are sources of error and limitations associated with the previously described mechanisms. When estimating throughput, all mechanisms start timers with a handshake. The best ones then subtract off some multiple of an estimate of latency, which they assume to be the amount of time required for the handshake, but which may actually be substantially different due to operating system CPU scheduling policies. Further, none of the passive monitoring techniques estimate latency, only throughput. The MSHN system requires, at a minimum, the knowledge of both of these.

Based on these observations, we sought a passive mechanism that would accurately estimate both bandwidth and latency without requiring the application programmer to implement monitoring code. In the remainder of this paper we describe such a mechanism.

3. A Passive Approach for Monitoring Network Performance

Before describing our domain-independent mechanism for passively obtaining accurate network performance estimates, we enumerate some of the challenges we faced in evolving such a mechanism.

3.1 Challenges

In order to avoid modifying either the operating system or the system libraries, we chose to apply a technique developed by Condor [LIVN95]. That is, we chose to implement an additional library that intercepts `read()` and `write()` calls and then

link applications' object code with that library.[2] This additional library will then be able to pre-process parameters of `read()` and `write()` and post-process the results. We will define this interception of system calls as **wrapping** system calls.

After identifying the mechanism required to implement "domain-independence" and "passive monitoring" we turned our attention to the problem of accurately estimating bandwidth and latency. In the remainder of this section we refer to a process that issues a `write()` call to write across the network as the "writer" and the process that issues the corresponding `read()` as the "reader." In this paper, we also assume that the reader and writer are using TCP.

In order to estimate latency, we must know when the writer writes the message, and when the reader's computer receives it. We face several problems in trying to accurately obtain these times. First, the reader does not know when the writer wrote the message to the network. Second, if we modify `write()` so that the writer appends its local time to the beginning of the message, we still must compensate for the clock offset between the reader and writer computers. Since these computers do not have synchronized clocks, we cannot directly compare the writer's send time to the reader's receive time. Finally, if the writer writes the message long before the reader is ready to read the message, the message will be buffered on the reader's machine. This makes it difficult to estimate the time of reception. We will discuss these problems in some depth after summarizing the corresponding problems associated with estimating throughput.

In estimating throughput, we face similar challenges. Because we do not have control over the operating system, we have difficulty estimating transmission time. This

[2] If object code is not available, techniques developed in Paradyn [PARA97] [LARU95] could be used to link this library with the executable.

affects our ability to estimate throughput. From an application's perspective, once it calls `read()`, it blocks and remains blocked until the operating system returns with data in the buffer. We could measure the total blocked time after an application makes a `read()` system call and assume that this elapsed time is an estimate of total transmission time. However, unless the `write()` that corresponds to the `read()` occurred at the same time, this assumption would most likely result in incorrect throughput estimates because of the composition of the blocked time.

We refer to two significant problems associated with estimating transmission time as the "early reader-late writer" problem and the "late reader-early writer" problem. The remainder of this section will further explain these problems.

In the "early reader-late writer" scenario, the reader calls `read()` and blocks waiting for the writer to execute `write()`. Some time after the writer finally writes, the reader receives the message and unblocks. In this case, the total blocked time is composed of both the time spent transmitting as well as the time spent waiting for the late writer. Because we cannot know how much of the blocked time was due to waiting for the late writer, we cannot assume that blocked time is equivalent to the transmission time. If we were to make such an assumption, we would underestimate throughput.

In the "late reader-early writer" scenario, the writer writes to the network on an established connection, but the reader has not yet called `read()`. The operating system on the reader's host may buffer some or all of the data received from the writer. When the reader finally calls `read()`, it reads the buffered portion of the message directly from local memory. In this case, the total blocked time is composed of time spent reading from local memory, as well as the time required to read the unbuffered portion of

the message from the network. In most systems, retrieving data from memory is significantly faster than network transmission time, so using this total time would result in an overestimate of throughput.

In summary, unless the `read()` and `write()` calls happen at exactly the same time, we cannot simply use blocked time to estimate throughput. Additionally, we face problems related to clock offset in estimating latency. Our approach to measuring network QoS will recognize and take advantage of the "early reader-late writer" scenario to aid in obtaining accurate latency and throughput estimates.

3.2 An Additional Observation That Helps

Many writes to the network by applications are large (e.g., files and graphics). As we saw in Netscape, a `read()` from the network returns immediately upon receiving data in the buffer. Even though an application writer specifies the amount to be read in the `read()` system call, the call will return with the amount of data actually read immediately upon receiving any data. To ensure that all desired data is read, the application writer must implement the application so that it repeatedly calls `read()` until it has read the entire message. We will use this observation in conjunction with the "early reader-late writer" scenario to help estimate throughput.

3.3 Our Passive Monitoring Approach

In this section we describe the passive network monitoring approach that we developed for MSHN. We will give an overview of our approach and then address the following four areas: clock offset compensation, the cooperating writer, the cooperating reader, and special considerations.

Overview of the MSHN Passive Network Monitoring Approach

Our approach exploits the "early reader-late writer" scenario. We have wrapped the `read()` and `write()` library calls to recognize TCP connections. In our approach, the `read()` system call recognizes the "early reader-late writer" scenario, allowing the estimation of end-to-end latency, and when appropriate, throughput.

In measuring end-to-end latency, we must address the three problems raised above. The first is that we do not know when the writer sent its message. We solve this by wrapping the `write()` system call to append, to the front of the message, a timestamp from the writer's clock. The second problem results from the fact that the reader's clock and the writer's clock are not synchronized, but the reader and writer need to have reference to a common timeline. We will address this problem in the next subsection. The final problem deals with the "late reader-early writer" scenario. We avoid this problem by detecting this situation and only calculating latency when we are sure that we are in the "early reader-late writer" scenario.

To estimate throughput, our mechanism must first estimate transmission time. In the "early reader-late writer" scenario, blocked time is composed of the end-to-end transmission time and the time spent waiting for the writer. We will exploit our observation that many network writes are large. As mentioned above, when a large message is read from the network, `read()` must be called repeatedly until the entire message is received. The two necessary components for calculating throughput are the number of bytes transmitted and transmission time for those bytes. Our algorithm computes the difference between the times of the first read and the last read. This difference is transmission time. Our technique "throws away" that first period of blocked time consisting of time due both to end-to-end data transmission as well as time spent

waiting for the writer. It is safe to assume that the remaining time that it takes to read the message is due primarily to transmitting the remainder of the message. The number of bytes received will be the message size less the size returned by the first `read()`. This allows us to estimate throughput between the reader and writer.

We use the term "cooperative" to refer to the reading and writing applications that are linked with our library. We discuss cooperative readers and writers in more detail shortly.

Compensating for Clock Offsets

Our passive network monitoring requires that the communicating hosts have access to a common time reference. Since we cannot assume that the member hosts of MSHN have perfectly synchronized clocks, we use a derivative of the Network Time Protocol (NTP) to estimate clock offsets between machines [COUL96]. Our approach is almost identical to the protocol as described in the reference, with the exception that we eliminate the need for one of the timestamps. As the reference does, we call the estimated clock offset o_i and the estimated error, d_i. In this protocol, shorter round trip times result in smaller errors. We exploit this observation by making multiple calls to the remote timeserver, and then keeping the offset that results from the shortest round trip time.

In MSHN, we would expect estimates of o_i and d_i to be available from the Resource Status Server (RSS), but prior to adding this functionality to the RSS, we implemented and tested our passive network performance monitoring algorithm by wrapping the `accept()` and `connect()` system calls to trigger these estimates. This extra code adds considerable overhead to these system calls. In the final implementation of MSHN the RSS would periodically, at times of low system usage, poll MSHN

members and record clock offset and drift. To minimize added network traffic for delivery, the distribution of o_i and d_i can be included with security certificates [WRIG98].

The Cooperating Writer

We incorporated, into the MSHN library, a wrapper for the `write()` system call that detects when `write()` has been called to write to a TCP connection. The wrapper appends the following information to the front of such messages: the writer machine's current time, $T_{remote'}$, and the size of the message.

The Cooperating Reader

Like the write() system call's wrapper, the read() system call's wrapper also detects when it is reading from a TCP connection. We now enumerate the steps that the wrapper takes if it detects that this is the first time that the `read()` is being called for the particular data set:

1. It records a local clock time stamp, $T_{blocked}$, prior to (possible) blocking.
2. When the `read()` continues (unblocks), the wrapped system call records the time, $T_{startRead}$.
3. $T_{remote'}$ and total message size are stripped from the first part of the message received.
4. $T_{remote'}$ is adjusted for clock offset between the reader and writer machines and this adjusted time is recorded as T_{remote}.
5. `read()` now tests to see whether the "early reader-late writer" situation has occurred. If $T_{blocked}$ occurred earlier than T_{remote}, then the reader was early. That is, the reader was blocked for a while waiting on the writer to write. Only in this case can latency be approximated.

   $$\text{Latency} = T_{startRead} - T_{remote}.$$

6. The received message data is passed to the application, with the return value of the `read()` system call decremented to account for the size of the timestamp and total message size fields.

We now describe the actions taken when the read wrapper is invoked after the initial `read()`.

1. The size of the message remaining is decremented by the amount of data in the buffer.

2. If the size of the message remaining is zero, the end of the message has been found. In this case, throughput can be calculated.

3. The `read()` wrapper calculates throughput using:
Throughput = (total message size – size of first part of message)/($T_{endRead}$ -$T_{startRead}$)

We note that throughput is only calculated when more than one read() is invoked to obtain the data that was sent. We also note that the throughput and latency estimates are estimates of end-to-end throughput, which are, in fact, what MSHN is interested in.

Special Consideration

Latency and throughput can only be calculated for a subset of the total network traffic, that is when the "early reader-late writer" scenario is true. We can modify the algorithm to increase the opportunities to estimate throughput by "loosening" the "early reader-late writer" requirement if we have a good estimate of the absolute minimum latency, $Latency_{min}$ between the communicating machines. Rather than requiring $T_{blocked}$ to occur before T_{remote} (that is, $T_{blocked} - T_{remote} < 0$), we could allow $T_{blocked}$ to be up to the minimum latency later than T_{remote} ($T_{blocked} - T_{remote} < Latency_{min}$). This seems insignificant for machines connected locally over high speed networks where latency is in the order of milliseconds or fractions of milliseconds. However, consider machines connected over extended network links, especially those using satellite communication. In this case, latency is in the order of 100's of milliseconds and this loosened requirement

could prove significant. In this latter case, the opportunities to estimate throughput could increase dramatically with this modification.

4. Summary

In our review of existing tools and application components, we classified the network monitoring techniques as either passive or active. Passive monitoring has the desirable attribute of minimal added overhead while active monitoring gives the ability to measure latency and is not tied to any particular application. We then presented an approach that makes use of the low overhead of passive monitoring, estimates end-to-end latency and throughput, and is independent of any particular application. Preliminary experiments show that our technique adds 6% to the required execution time of the `read()` system call, confirming the low overhead of passive monitoring. We anticipate that using the tools developed by Oregon Graduate Institute's Synthetix team could further reduce this overhead [PUCA96].

We plan further work to quantitatively assess the accuracy of predictions based upon our mechanism.

5. Acknowledgements

We gratefully acknowledge Professor Cynthia Irvine's suggestion that we use the number of bytes to be sent rather than start and end flags. Although this solution is not guaranteed to work with every implementation of write, when it does work, it certainly reduces the overhead that we encounter when using start and end flags and bit stuffing.

6. References

[COUL96] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems, Concepts and Designs, 2d Edition*, Addison-Wesley, NY, 1996.

[BERK91] Berkeley Unix Distribution, *Unix Man Pages*, March 1991.

[HAYE94] G. Hayes-Roth, and L. Erman, *The Joint Task Force Architecture Specification (JTFAS)*, Teknowledge Federal Systems, Palo Alto, CA, 1994.

[HEWL96] Information Networks Division, Hewlett-Packard Company, *Netperf: A Network Performance Benchmark, Revision 2.1*, February 1996.

[KIDD96] T. Kidd, D. Hensgen, R. Freund, L. Moore, "SmartNet: A Scheduling Framework for Heterogeneous Computing", *ISPAN*, 1996.

[KRES97] J. Kresho, *Quality Network Load Information Improves Performance of Adaptive Applications*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1997.

[KRES98] J. Kresho, D. Hensgen, T. Kidd, and G. Xie, *Determining the Accuracy Required in Resource Load Prediction to Successfully Support Application Agility*, EURO-PDS98, 1998.

[LARU95] J. Larus and E. Schnarr, *EEL:Machine-Independent Executable Editing*, SIGPLAN PLDI 1995.

[LEEC98] C. Lee, J. Stepanek, B. Michel, I. Foster, C. Kesselman, R. Lindell, S. Hwang, J. Bannister, and A. Roy, *Qualis: the Quality of Service Component for the Globus Metacomputing System*, IWQoS98, Napa, CA, 1998.

[LIVN95] M. Livny, M. Litzkow, T. Tannenbaum, and J. Basney, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Dr Dobbs Journal, February 1995.

[NETS98] Netscape Communication's Corporation, *Netscape Communicator Source Code*, March 1998.

[PARA97] Paradyn Project, *Paradyn Parallel Performance Tools User's Guide Release 2.0*, University of Wisconsin-Madison, 1997.

[PERK97] D. Perkins, and E. McGinnis, *Understanding SNMP MIBs*, Prentice-Hall, NJ, 1997.

[PUCA96] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang, *Optimistic Incremental Specialization: Streamlining a Commercial Operating System*, Oregon Graduate Institute, 1996.

[SILB98] A. Silberschatz, and P. Bae Galvin, *Operating Systems Concepts, 5th Edition*, Addison-Wesley, Menlo Park, CA, 1998.

[SPRI97] N. Spring, *Network Weather Service for Mentat 3.0 User's Guide*, October 1997.

[STAL98] W. Stallings, *Cryptography and Network Security, Principles and Practice, 2nd Edition*, Prentice Hall, Upper Saddle River, NJ, 1998.

[WOLS97] R. Wolski, N. Spring, and C. Peterson, *Implementing a Performance Fore-casting system for Metacomputing: The Network Weather Service*, SC97 Technical Paper, 1997.

[WRIG98] R. Wright, D. Shifflett, and C. Irvine, *Security for a Virtual Heterogeneous Machine*, to appear in Proceedings of the 12th CSA Conference, Scottsdale, AZ, 1998.