A SMART TCP SOCKET FOR DISTRIBUTED COMPUTING

SHAO TAO

SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2004

Name: Shao Tao
Degree: B.Sc.(2nd Upper Hons.)
Dept: School Of Computing
Thesis Title: A Smart TCP Socket for Distributed Computing

Abstract

Middle-ware in distributed computing coordinates a group of servers to accomplish a resource intensive task; however, the server selection schemes without resource monitoring are not yet sophisticated enough to provide satisfying results at all time. This thesis presents a Smart TCP socket library using server status reports to improve selection techniques. Users are able to specify the server requirements by using a predefined meta language. Monitoring components such as the server probes and monitors will be in charge of collecting the server status, network metrics and performing security verifications. A user request handler called *wizard* will make the best match according to the user request and the available server resources. Both centralized and distributed modes are provided so that the socket library can be adapted to both small distributed systems and a large scale GRID. The new socket layer is an attempt to influence changes in the middle-ware design. It allows multiple middle-ware implementations to co-exist without introducing extra server load and network traffic. Thus, it enables middle-ware designers to focus on improving the task distribution function and encourages the popularity of GRID computing facilities.

Keywords: TCP Socket, Middle-ware, Bandwidth Measurement
Server Selection Technique, Active Probing, Resource Monitoring
Lexical and Syntactical Analysis

A SMART TCP SOCKET FOR DISTRIBUTED COMPUTING

SHAO TAO
(B.Sc(2nd Upper Hons), NUS)

A THESIS SUBMITTED FOR THE DEGREE OF
MASTER OF SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgements

It has been six years since the first day when I came to NUS. I received enormous help and support from my family, my supervisor and many friends around.

I have to thank my family for their encouragements through the years. My father told me to always be an honest man. My mother supports me to pursue higher academic achievements. And my brother shares the joy and sadness with me.

My deepest thanks to my supervisor Prof. Ananda, for guiding me through my honors year, now my master project and giving me a chance to teach in the school. Prof. Ananda has provided insightful new ideas to this master topic and leads me to the correct research direction when I was confused from time to time.

Kind thanks to the friends and school mates around me for spending the after-school days together and making my life here enjoyable.

Contents

Summary

Middle-ware in distributed computing coordinates a group of servers to accomplish a resource intensive task. To accommodate various applications, certain servers with particular resource usage feature and configuration will be more preferable than others. Without resource monitoring, the server selection techniques are mainly based on static configuration statements manually prepared or random process such as round-robin function. These rigid techniques cannot precisely evaluate the actual running status of servers. Thus, they are not able to provide the optimal server group.

In this thesis, a Smart TCP socket library using server status reports to improve selection techniques is presented. The library provides a meta language for describing server requirements. With the rich set of parameters and predefined functions, users can write highly sophisticated expressions. It also provides a convenient client library which can be used stand alone or combined with other libraries for better performance. The library's a flexible structure, that enables developers to plug in new components or upgrade existing ones conveniently. Both centralized and distributed modes are available so that the socket library can be adapted to both small distributed systems and a large scale GRID.

List of Tables

List of Figures

List of Abbreviations

API Application Programming Interface

ASCII American Standard Code for Information Interchange

BSD Berkeley Software Distribution

BW Bandwidth

ICMP Internet Control Message Protocol

IO Input/Output

IP Internet Protocol

IPC Interprocess Communication

ISN Initial Sequence Number

LVS Linux Virtual Server

M_{net} Network Monitor

MPI Message Passing Interface

M_{sec} Security Monitor

M_{sys} System Monitor

MTU Maximum Transfer Unit

NAC Network Admission Control

NAT Network Address Translation

NMAP Network Mapper

OS Operating System

PVM Parallel Virtual Machine

Req/Rep Request/Reply

RTT Round Trip Time

Seq Num Sequence Number

SLoPS Self-Loading Periodic Streams

TCP Transmission Control Protocol

UDP User Datagram Protocol

List of Publications

1. "A TCP Socket Buffer Auto-tuning Daemon", Shao Tao, L. Jacob, A. L. Ananda. ICCCN 2003, Dallas TX USA, 2003.

2. "A Smart TCP Socket for Distributed Computing", Shao Tao, A. L. Ananda. to appear in ICPP-2005, Oslo Norway.

Chapter 1

Introduction

In this chapter, we will introduce the motivation behind this project and some background information. The objectives of the project will be explained later, followed by an outline of the remaining chapters.

1.1 Motivation

The TCP socket library provides a rich set of APIs for users to easily build up network applications. Its availability in many operating systems enables network applications to communicate with one another, even when running on different architectures. With the growth of distributed programs on the network, the traditional socket library shows a few limitations in functionality that can be improved. Most distributed computation applications like graphic rendering, gene sequence analysis and cryptography calculation, consider networked servers as an abstracted grouped computation service accessible through sockets.

Within a controlled computation network, where servers providing identical services are monitored, it is redundant for an application to specify the names of the servers to use, as shown in Fig. 1.1. Also, a particular server referenced by the server name may not be available at a particular moment. A recovery mechanism must be established for such a case in order to make use of alternative servers.

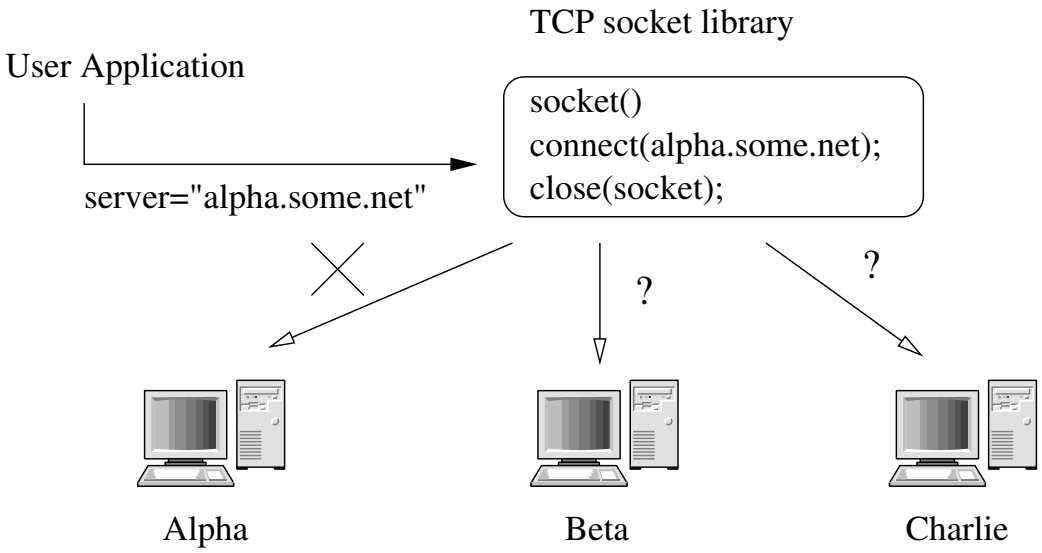


Figure 1.1: Resource Referred by Server Name

Distributed applications normally involve large amount of read and write operations over multiple sockets. The standard socket library does not provide convenient interfaces for creating and closing a group of sockets. When multiple servers are required, the same sequence of function calls are made multiple times for creating each new socket as depicted in Fig. 1.2. For such applications, a wrapper socket function would be preferable than duplicating code segments. Instead of returning a single socket, the wrapper function returns a list of sockets that will participate in a single computation

task. The functions implemented over these sockets are determined by the programming paradigm and algorithm.

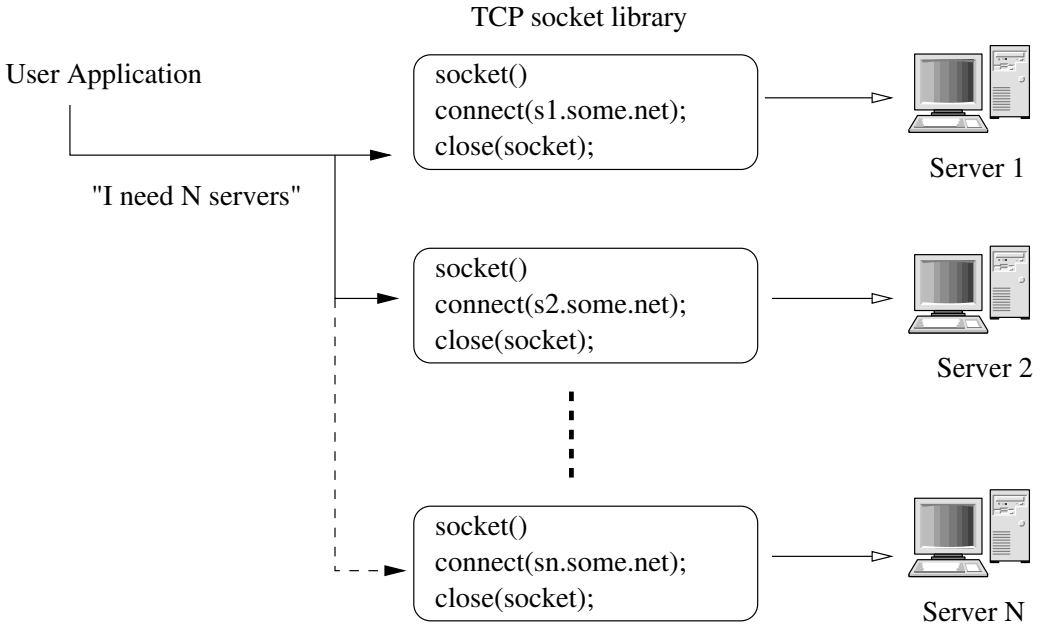


Figure 1.2: Request for Multiple Sockets

Fig. 1.3 reveals another limitation of the standard TCP socket library - users have no methods to specify the requirement for the servers. In a cluster of servers targeting on the same task, performance may vary due to the system configuration or current workload of servers. Applications may have different requirements for various system resources at different intensity levels. A memory intensive program should be run on machines with sufficient amount of free memory space. A data intensive program would achieve better performance on servers with less hard disk Input/Output activities and network load. An interface is necessary to inform socket library about the server qualification standard for a particular application.

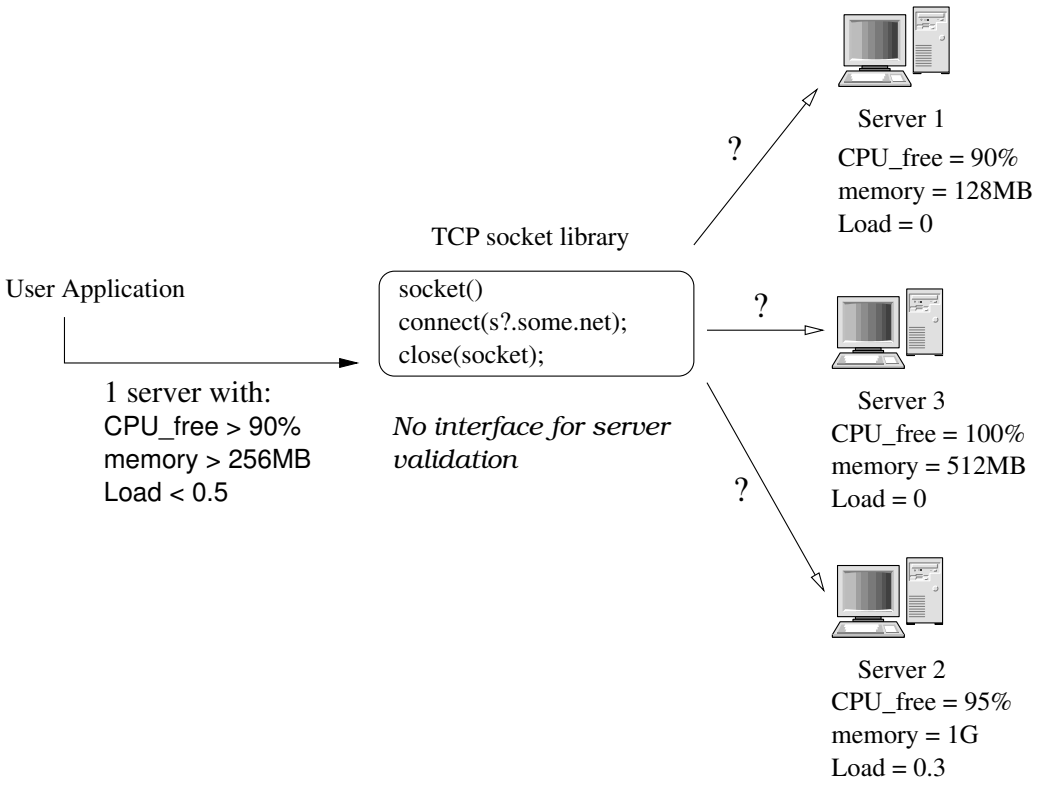


Figure 1.3: User Requirements for Servers

1.2 Background

Abundant amount of research works have been done to improve distributed and parallel programming environments, including message passing libraries, independent task schedulers, frameworks for large scale resource management and system patches for automatic process migration at kernel level. A list of these utilities is shown in Table. 1.1.

The message passing library allows users to develop distributed applications with the convenient function calls for passing messages and data structures among the computational nodes. PVM[pvm04], MPI[mpi04] and P4[p4system93] libraries belong to this category. The task schedulers like Ants[ants04], *Condor*[condor04] and *Linux Virtual Server*[lvserver04] are independent programs focusing on redistributing users' tasks among multiple servers according to the deployed load balancing algorithms used. The *Condor* tool set allows users to give *Classified Advertisement*[rajesh98] to describe their job properties and assigns the task to matched servers.

The *Globus* project[globus04] provides a framework to standardize the representation of services and system resources in order to provide uniform interfaces for service publication/discovery, resource management and efficient data exchange. Another category contains the patches for automatic system load balancing at system level. OpenMosix[openmosix04] project belongs to this category, available for Linux kernel 2.2 and 2.4 series under the i386 architecture. It requires the program at application level to use the *fork()* system call to create multiple processes at run time.

The Smart socket library in this project is an approach in programming

Name	Type	Description
PVM, MPI, P4	programming library	message passing for application
Smart library	programming library focusing on network layer	server selection by user
LVS, Ants, Condor	Task scheduler	tasks distribution among servers
Globus	framework for GRID service	format definition of service
OpenMosix	kernel patch	automatic process migration

Table 1.1: Current Distributed Programming Tools

library category. Instead of providing convenient message passing interfaces for application, we focus on the network layer and provide interfaces for users to state the characteristics of the servers desirable for their applications.

1.3 Objectives

The new Smart socket library is designed with the following objectives for sever selection in a scalable distributed environment:

- The workload status of servers should be extracted with low overhead.
- There should be an organized format to present the user's requirement for server resources.
- Users can easily employ the new socket library in a small scale local computation environment and a large scale environment with numerous servers scattered globally.
- The convenient socket library interface must be provided for easy application development.

- The structures of the components must be flexible in order for future enhancements as well as cooperating with other distributed facilities.

1.4 Thesis Contribution

This project has made the following contributions:

- A basic structure for status-aware server selection at application level is implemented. The status information can be extracted from operating system interface, the network monitors or security agents.

- A meta language for describing user's requirement on servers is implemented. With the abundant build-in parameters and mathematical functions, user are able to write complex representations conforming to sophisticated algorithms.

- The convenient client library can be used, stand alone or combined with other tools such as the PVM library, complementing the deficiencies of the existing utilities.

- The server probes, network monitors, security agents and the server selection algorithms used by the wizard program can be replaced conveniently as long as the information messages conform to the predefined format.

- A distributed mode matrix multiplication program and a concurrent downloading program have been developed to verify the applicability of this library.

With the Smart socket library, users can explicitly select servers for the applications. An example is given in Fig. 1.4, in which a user requests for 3 servers. Each server must have 100 MBytes free memory and the CPU usage must be less than 10%. Also, the network delay to each server should be less than 20 ms and the host named "hacker.some.net" must not be selected. There are 12 available servers located in four networks: A, B, C and D, with a network delay of 100 ms, 5 ms, 10 ms and 15 ms each. The *wizard* program scans through each network sequentially for candidates. All servers in network A are eliminated due to the long network delay. Host B2, C1 and D1 are qualified based on the requirements. Host C2 is not chosen since it is blacklisted.

1.5 Thesis Outline

Chapter 2 will present the related works done by other researchers. Chapter 3 will introduce the design issues and key components in the project. In Chapter 4, we will discuss some of the implementation issues. The experiment results will be shown in Chapter 5 to verify the effectiveness and applicability of this project. The limitations and future work of the project are described in Chapter 6, followed by the conclusion in Chapter 7.

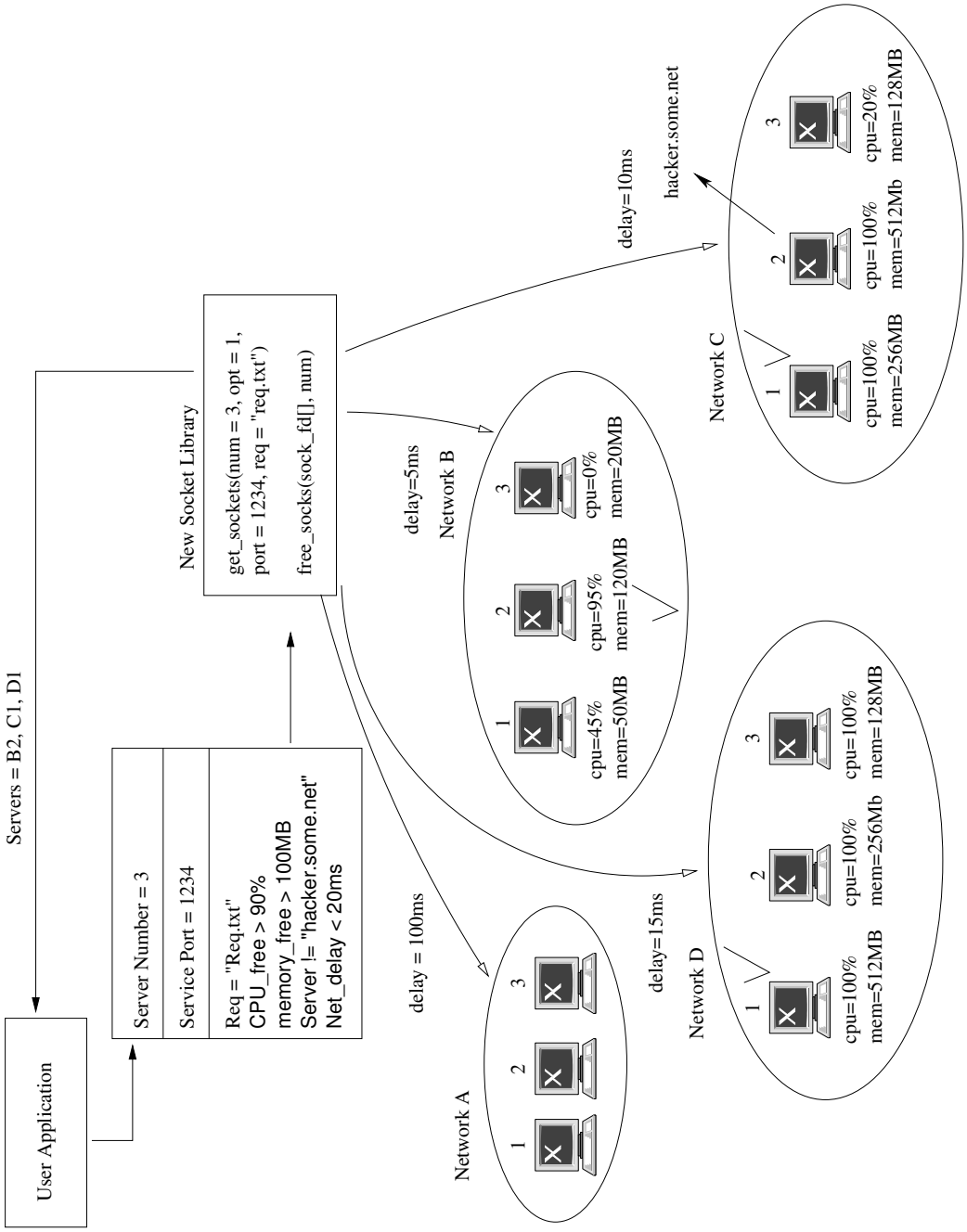


Figure 1.4: An Example with Smart Socket Library

Chapter 2

Related Works

This project involves several aspects of distributed computing, such as resource monitoring, programming interface development and user query handling. In this chapter, we will present some related projects and the comparisons between our project and these previous works.

2.1 Status Report

The `/proc` file system[erik01] in Linux system is used to extract the system parameters from the servers. It provides access to system information about the hardware devices like CPU, memory, network interface and hard disk. Device drivers and kernel modules can create corresponding entries in `/proc` for providing device information or debugging purposes. It is an efficient way for kernel level information retrieval in the Linux system.

The *Trust Agent* from Cisco Systems is a probing agent running on the ending host. It interacts with the softwares in the local host to report infor-

mation like system version, patch level and computer virus infection records. Currently, Cisco Trust Agent supports only Windows systems. The server probe developed in this project supports only Linux systems due to the dependence on *procfs*. However, based on the simple message passing mechanism, a windows agent can be quickly built based on Windows APIs.

The system probe in the Smart TCP socket library is similar to the Cisco trust agent. It is installed in each server being monitored and performs active self-probing periodically. The system resource usage is extracted from `/proc` entries, written into a server status report and sent back to the system monitor.

For the network metrics measurement, the network delay and available bandwidth are critical for this project. Numerous popular tools are available to the public for bandwidth estimation, including *pipechar*[ncs03] and *pathload*[manish02pl]. *Pathload* uses an end-to-end technique containing a sender and a receiver. The sender transmits multiple data streams with different data rate, following which the arriving time of the data packets are recorded. If the transmission time dramatically increases, after transmission rate exceeds a certain value, that value will be used as the estimated available bandwidth. *Pathload* is a two-end probing technique, which needs the sender and receiver programs running on both ends of the target network link. This technique is highly accurate but less flexible compared with the single end probing techniques.

Pipechar developed by Lawrence Berkeley National Laboratory is an one-end probing technique. It uses the packet pair method to estimate the link capacity and bandwidth usage. It sends out two probing packets and mea-

sures the echo time. The bandwidth value is calculated based on the gap in the echo time. As a single end packet pair based tool, *pipechar* is very flexible but less robust to network delay fluctuations.

The Smart socket library uses an one-end probing technique derived from the packet pair method, named one way UDP stream, to probe the target network link. The differences between probing packet sizes and delays are used to estimate the available bandwidth.

2.2 Distributed Computing Libraries

MPI(Message Passing Interface Standard)[mpi04] and PVM(Parallel Virtual Machine)[pvm04] are the two common libraries available for distributed application development. MPI is a standard defining a set of application programming interfaces for efficient communication in heterogeneous environment. There is no virtual server or resource management ideas concerned in the original design. Users must use the communication functions in the MPI implementation to coordinate the processes in the applications.

PVM is an application library that enables the user program to spawn multiple processes in a cluster of servers and provides inter-communication among these processes. The design of PVM is based on the concept of *virtual machine*. It includes programming interfaces for exchanging different types of data, managing the spawned processes and controlling the servers used by the current program. The user can manually monitor or manage the machines through the PVM console and the applications can modify the server pool at run time. A detailed comparison between MPI and PVM is given in [geist96].

MPI and PVM are application level libraries focusing on message passing and process management. The client library in the Smart TCP socket library enhances the network layer functions, focusing on server selection and socket management according to the user's requirement. As the Smart library is working at a different layer compared with many other distributed libraries, it has a great compatibility, which allows users to apply other distributed libraries such as PVM and the Smart library in the same application.

2.3 Grid Middle-ware

The Globus Alliance project[globus04] started with a goal of "enabling the application of Grid concepts to scientific and engineering computing". The Globus project provides the Globus Toolkit for quick building Grids and Grid applications. This toolkit contains a group of components: Globus Resource Allocation Manager for resource and process management, Globus Security Infrastructure for user authentication service, Monitoring Discovery Service(MDS) for accessing system configuration, network datasets, and Heart Beat Monitors for detecting system failure. The Globus project presents a new layering of network based on the resource sharing concept for scientific computing[ogsa04]. The Globus Grid Architecture contains the following layers: Application, Collective, Resource, Connectivity and Fabric. According to this new network layering, our project is working at the connectivity and resource layer, which focuses on providing better computation resources for user tasks.

The resource monitoring function in the Smart socket library is similar

to the MDS component in Globus toolkit. Globus toolkit provides interfaces for applications to access the resource information. The Smart socket library manages the resource information internally, hides the lower layer details from users and provides clean programming interfaces for application development. The objectives for resource monitoring in the Globus toolkit and the Smart socket library are different. The Globus toolkit tends to provide users an overview of the resources in the GRID environment. The Smart socket library automatically monitors the resources, minimizes the user's involvement and tries to provide the optimal resource for the upper level applications.

The Condor[condor04] project developed a set of utilities for providing services like resource monitoring, task planning/scheduling and process migration. The user level applications need not be modified in order to use the Condor tool set. The Condor tool set provides Classified Advertisement(*classad*) for users to specify server requirements and for servers to specify the backward requirements on users' tasks. The matchmaker will try to pick the best resources for that matched task. Another great feature provided by Condor is that it provides process migration, which is used when one part of the task cannot be finished on a particular server in time. Though both the *classad* from the Condor project and the meta language defined in the Smart socket library can be used to describe the requirements on server resources and network metrics, there are some differences. In *classad*, different types of parameters may be defined including numerical type, character string type and so on. Users can specify the requirements on the server resources; meanwhile, servers can also define the characteristics of the user tasks that can be run locally. The query handler called *match-maker*

allocates the best matched servers for each task. The meta language in the Smart socket library provides mainly numerical type parameters. It covers a larger parameter range, from system load, CPU usage, disk input/output activities to network metrics. A set of predefined mathematical functions are available, which can be used to give complicated requirement specifications if necessary.

2.4 Load Balancing Tools

Although load balancing is not a major concern for this project, it could be considered as an advanced feature for future development.

The Linux Virtual Server[lvserver04] is a utility running in the gateway of a server cluster. It accepts the application request and forwards the request to the servers running behind the gateway. The decision making could be based on round-robin, Hash function or accounting information like number of tasks completed by each server or number of connections currently established to the servers. This utility has been included in the new Linux kernel 2.6 series.

OpenMosix[openmosix04] has a very different way to parallelize applications compared with the other distributed application tools. OpenMosix modifies the Linux kernel to add the daemons inside. The Linux systems with OpenMosix patch communicate with each other and build a cluster automatically. If the user application can create multiple processes during execution, some of the processes will migrate to run on other servers in the cluster. In future work, the Smart socket library can be modified to provide abstract socket interfaces for process involving network communication, such

that the process suspension/resumption and data migration can be realized smoothly.

Chapter 3

Components and Structure

In this section, the key components of the Smart socket library and the bandwidth measurement method will be presented in detail.

3.1 Overall Structure

The Smart TCP socket library contains 7 components. The overall structure diagram is given in Fig. 3.1. *Server probes* are running on the servers in the computing environment. *System monitor*, *network monitor* and *security monitor* run on the monitor machine. On the same monitor machine, there will be a *transmitter* to send the information collected by the monitors to the wizard machine. On the wizard machine, we have *wizard* program and *receiver* program running. *Receiver* writes the message received from the *transmitter* to the memory space shared with *wizard*. *Wizard* will wait for user's request from the client machine and process it using the status reports. An insight view of these components will be given in the rest of this chapter.

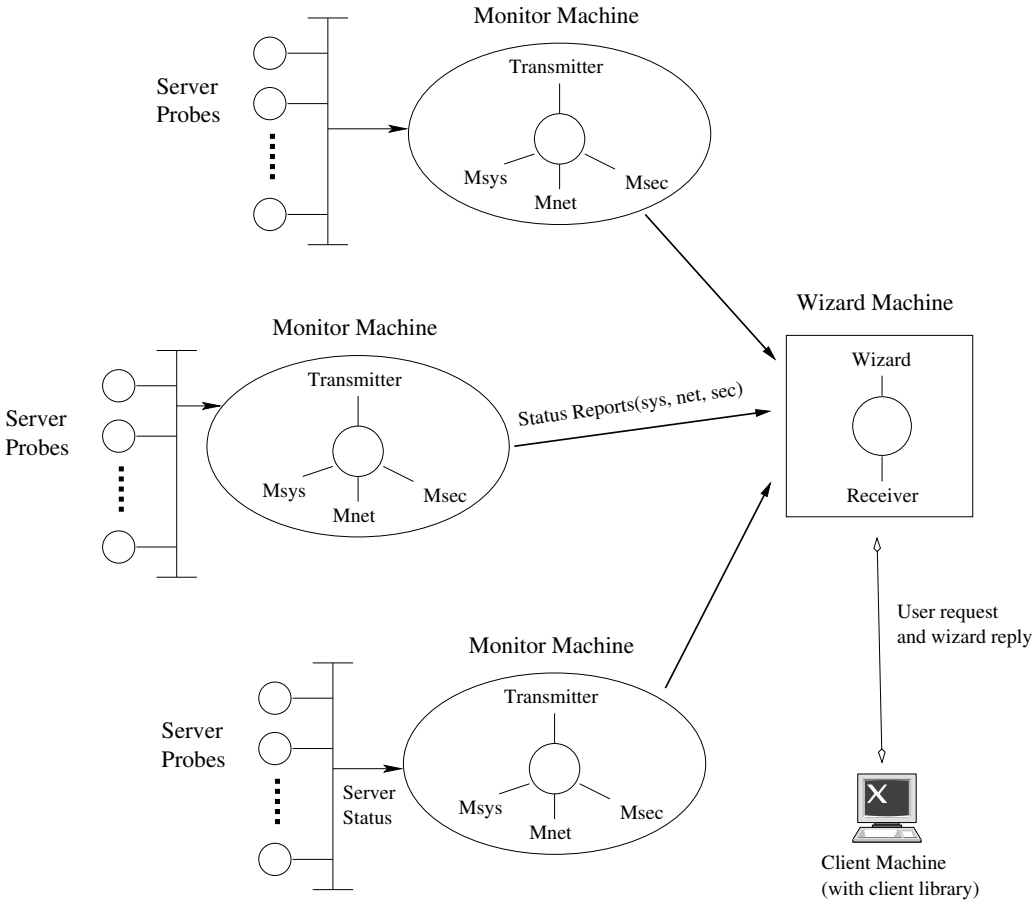


Figure 3.1: Overall Structure of the Smart TCP library

3.2 Server Probe and Status Monitor

3.2.1 Server Probe

The *proc* file system in Linux provides a convenient way for users to access the system information, such as settings of devices, hardware configurations and the system resource usage. The server resource usage status includes the following critical parameters in Table. 3.1.

Entries	File	Meaning
load_1, load_5, load_15	/proc/loadavg	system load in 1, 5, 15 minutes
user, nice, system, idle	/proc/stat	CPU usage rate
total, used, free	/proc/meminfo	memory usage
allreq, rreq, rblocks wreq, wblocks	/proc/stat	disk IO
name, rbytes, rackets tbytes, tpackets	/proc/net/dev	network interface IO

Table 3.1: Server Status Entries

The `/proc` entries will be scanned regularly and the scanned results will be sent back to the server status monitor - *system monitor*. The monitored parameters are selected to facilitate different types of applications: CPU bound, memory bound and IO bound. Large calculation tasks may require more CPU time and tremendous amount of free memory space. Data transmission tasks will prefer those servers with more network bandwidth and low disk read/write activities.

Once the status is collected, the server probes will send the status report to the system monitor running on a dedicated server. As the system monitor running in the local network has the minimal network delay and very few packet losses, the transport layer protocol in use is UDP in order to reduce

the overhead of the probing. If more parameters are required from the server probes, the size of server reports could increase dramatically. In that case, the reliability of the TCP protocol is preferable over the efficiency of UDP.

Currently every server probe requires 130 KBytes of memory space and the CPU usage is less than 0.2% on a Pentium-3 866 MHz machine. The server status report message is less than 200 bytes long. With a probing interval of 5 seconds, the required network bandwidth for status reporting from a single server is less than 40 bytes/sec. The server status report parameters are formatted into a character string for transmission. For example, if the network interface has a data transmission throughput of 200,000 bytes/sec. It will be transmitted as a string of "2000000", 7 characters long. In binary format, this number could be represented as an Integer type, typically 4 bytes long. Transmitting numbers as strings will require larger memory than what the actual figures would require in binary format. However, the advantage is that the probes can run on both machines with Big Endian(IBM, Motorola) and Little Endian(VAX, x86), without any modification, as the there is no memory alignment issue in transmitting character strings on networks.

3.2.2 System Status Monitor

The system status monitor receives the server status reports from system probes and writes them into the shared memory space, as demonstrated in Fig. 3.2.

The status reports are transmitted at an interval set by the administrator, normally 5 to 10 seconds. Once a report is received, the system monitor will

compare the server's address with the records in the shared memory space. If the server's address already exists, the original record will be updated with the new data. Otherwise, a new server record will be inserted into the server status database.

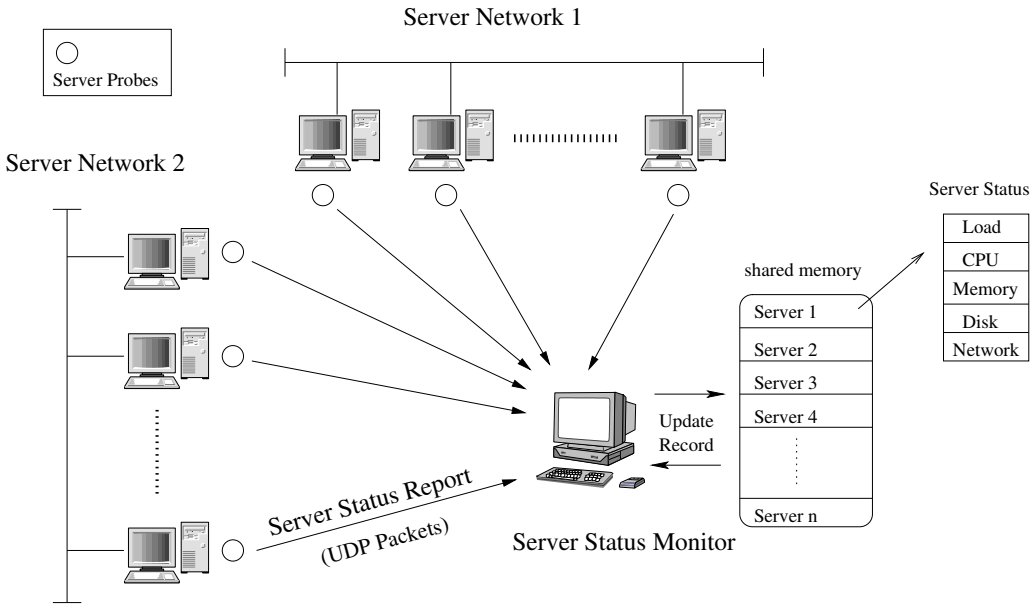


Figure 3.2: The relation between Server Probe and Monitor

A report timer is maintained in the system monitor side and each server status record in the status database is tagged with the time stamp showing when the record was recently updated. The monitor scans through the status database accordingly to remove the stale records regularly. This allows servers to join and leave the distributed environment at any time. If the server probe stops sending back the server report, the monitor will conclude that the server is not participating in any computation tasks. No more task will be assigned to that expired server, until the server probe resumes.

The server status database in the shared memory is also accessed by the

transmitter, which will transfer the status database to the wizard machine. To allow concurrent access and avoid conflicts, System V IPC mechanisms are used. The combination usage of System V semaphores and shared memory will resolve the concurrent read/write conflict situation and avoid false memory access problems.

3.3 Network Monitor

3.3.1 Network Metrics Measurements

The network metrics involved include the network delay and the available bandwidth of a network path. The packet loss rate is relatively low under today's high speed networking technology.

A lot of previous works have been done on measuring the network available bandwidth, including *nettest*, *iperf*, *pipechar* and *pathload*. *Nettest* and *iperf* are built based on path flooding method. Pipechar makes use of packet chain method and pathload uses the Self-Loading Periodic Streams(SLoPS) method. Nettest and Iperf uses end-to-end method: the sender program sends a TCP/UDP stream of packets as fast as possible and the receiver measures the receiving rate of the packets as the available bandwidth along the network path. This method is intrusive as it imposes heavy workload on the probed network. Pipechar sends a chain of UDP packets back to back and uses ICMP error messages to measure the gap created by the bottleneck network links. On network paths with a high delay variation, the estimated results will be inaccurate. Pathload uses a non-intrusive method

called SLoPS. The basic idea of SLoPS is to send streams of UDP packets at different data rate and monitor the network delay for each stream. If the sending rate is higher than the available bandwidth on the network path, the delay will be increased as the queue will be built up at the bottle link. According to our experiments, pipechar and pathload produce most accurate results. However, for networks under heavy load or with high delay variations, pipechar will report wrong results, because its probing algorithm is highly sensitive to network delay variations.

A modified one-way UDP stream method is used to measure the bandwidth and network delay in our project. We will take a look at this method in the following section.

3.3.2 One Way UDP Stream Measurements

The bandwidth measurement method used in this thesis, called one way UDP stream method, is a derivation of packet pair dispersion technique[carter96]. It does not require end to end connection to be established. Only the sender is responsible for sending the probing packets and measuring the network statistics. The advantage is the flexibility, although the result may not be as accurate to the end to end methods used by some network bandwidth measurement tools.

The main idea behind this method is that the network delay for transmitting a particular amount of data is related to the available bandwidth at that moment, which can be represented by the following formula:

$$\texttt{Network Delay} = \frac{\texttt{Data Size}}{\texttt{Available Bandwidth}} \tag{3.1}$$

However due to the measuring technique in the programs, the measured *Network Delay* is also affected by the system overhead and some network delay factors unrelated to the amount of data transmitted in the probing. In that case, the simplified version of the bandwidth formula(3.1) should be further extended to Formula(3.2)

$$\begin{aligned}\texttt{Network Delay} \;=\; & \frac{\texttt{Data Size}}{\texttt{Available Bandwidth}} + \\ & \texttt{System Overhead} + \texttt{Network Overhead}\end{aligned} \tag{3.2}$$

From *Computer Networking*[kurose03], the network delay for a a packet in a packet switch network is contributed by 4 factors as shown in Equation(3.3).

$$d_{delay} = d_{proc} + d_{trans} + d_{prop} + d_{queue} \tag{3.3}$$

In Equation(3.3), the *Network Delay* is a combination of *Processing Delay* - time to determine packet forwarding path, *Transmission Delay* - time for transmitting the data from host/router to the network link, *Propagation Delay* - time for the data bits to propagate from one end of the network link to the other end and *Queuing Delay* - time that data bytes have to wait in the router's queue. *Processing Delay* is determined by the packet size and processing speed of the networking device. *Propagation Delay* is determined by the network link distance and the signal propagation speed[steve01] in

the transmission medium. *Queuing Delay* is related with the amount of cross traffic along the network path and routers' scheduling algorithms.

Equation(3.3) contains the network delay factors related with system processing speed, the data size and the cross traffic. *Processing Delay* and *Propagation Delay* are usually negligible as the processing speed of the network device is fast and propagation speed of signal is rather high. The two dominating factors are d_{trans} and d_{queue}. In a simplified model, assuming S is the size of the data, R is the transmission rate of the network path and Q is queue length, we have:

$$d_{trans} = \frac{S}{R},\ d_{queue} = \frac{Q}{R}$$

So let T be the network delay to transmit data of size S, we can derive that:

$$T = d_{trans} + d_{queue} = \frac{S}{R} + \frac{Q}{R}$$

If we divide data size S by network delay T, we get:

$$B = \frac{S}{T} = \frac{S}{\frac{S}{R} + \frac{Q}{R}} = \frac{S}{S+Q}R$$

The result B can be considered as the available bandwidth to the data stream for transmitting S, which is proportional to the ratio between data size S and the queue length Q. In the actual scenario, we may consider the network path as a multi-hop route, where the narrow link and bottle link may not necessarily be the same and the queue lengths at routers vary

from time to time. In such cases, we need to measure the network delay and packet loss precisely and an end-to-end measurement method is preferred. A sophisticated model has been presented in Manish's paper[manish02]. In this thesis, due to the consideration about the flexibility, the simple model is used to serve the purpose.

As the network delay we measure contains the overhead from system and network, we can further improve the representation of network delay T to be:

$$T = \frac{S}{B} + Overhead_{sys} + Overhead_{net} \tag{3.4}$$

According to Equation(3.4), the overhead in network delay will affect the estimated value of available bandwidth B. In our algorithm, we send out two data streams with different sizes S_1 and S_2 and measure the network delays as T_1 and T_2. We will have

$$\begin{aligned} T_1 &= \frac{S_1}{B} + Overhead_{sys} + Overhead_{net} \\ T_2 &= \frac{S_2}{B} + Overhead_{sys} + Overhead_{net} \end{aligned}$$

That implies Formula(3.5)

$$B = \frac{S_2 - S_1}{T_2 - T_1} \tag{3.5}$$

The available bandwidth measurement Equation(3.5) has been tested to be effective in a previous work[shaotao03]. However, the delay Equation(3.4)

cannot be used to explain the network delays we measured in certain situation, which will be described below.

A program was written to send out a series of UDP probe packets of different sizes and receive the ICMP *port unreachable* error message returned. The time between the moment we send out the UDP packet and the moment we receive the ICMP error message is recorded as the network delay for that UDP probing packet. We start from UDP packet with 1 byte in data payload part and increase the UDP payload size until 6000 bytes with a step size equal to 10 bytes in order for a high resolution.

One experiment was conducted between two machines in campus network, *sagit* and *suna*. The result graph of Round Trip Time(RTT) over UDP packet size is given in Fig. 3.3.

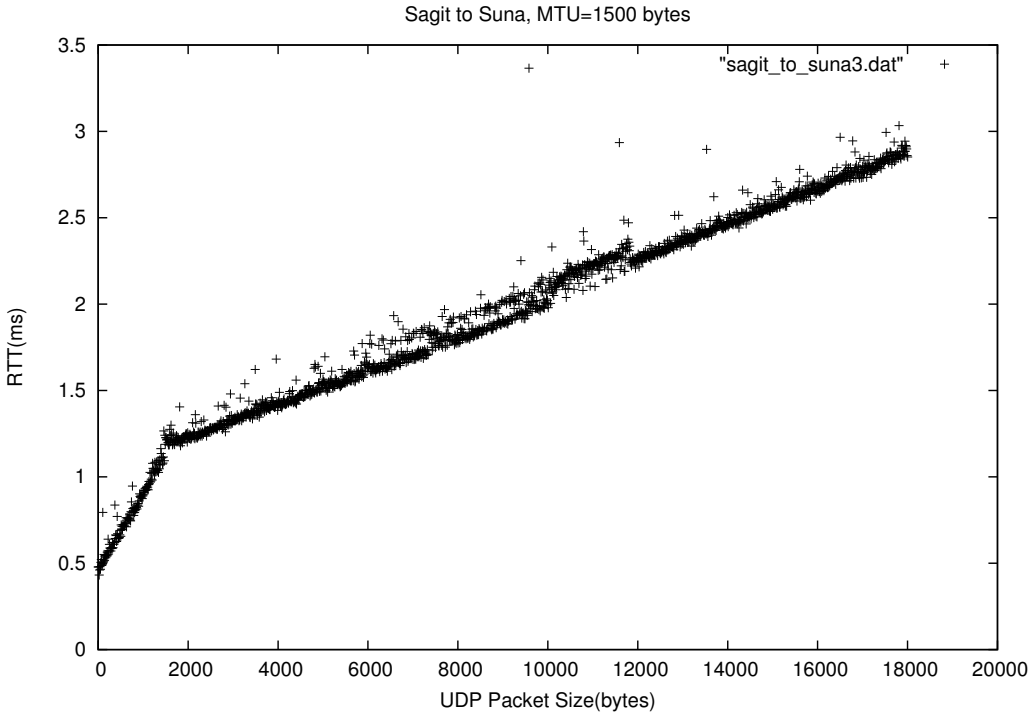


Figure 3.3: Round Trip Time from sagit to suna, MTU=1500 Bytes

We find that the Round Trip Time of the probing packets is not linearly proportional to the UDP packet size. Instead, there is a threshold point for the increasing packet size. The increasing rate of the round trip time is much higher when the UDP packet size is below the threshold. We further notice that the threshold is very close to the *Maximum Transfer Unit*(MTU) value. To verify this, another two sets of the same probing experiments were done from host *sagit* to host *suna*, the plotted graphs are given as Fig. 3.4 and Fig.3.5.

In Fig. 3.4, when the MTU value of the network interface was set to be 1000 bytes, the threshold appeared around 1000 bytes.

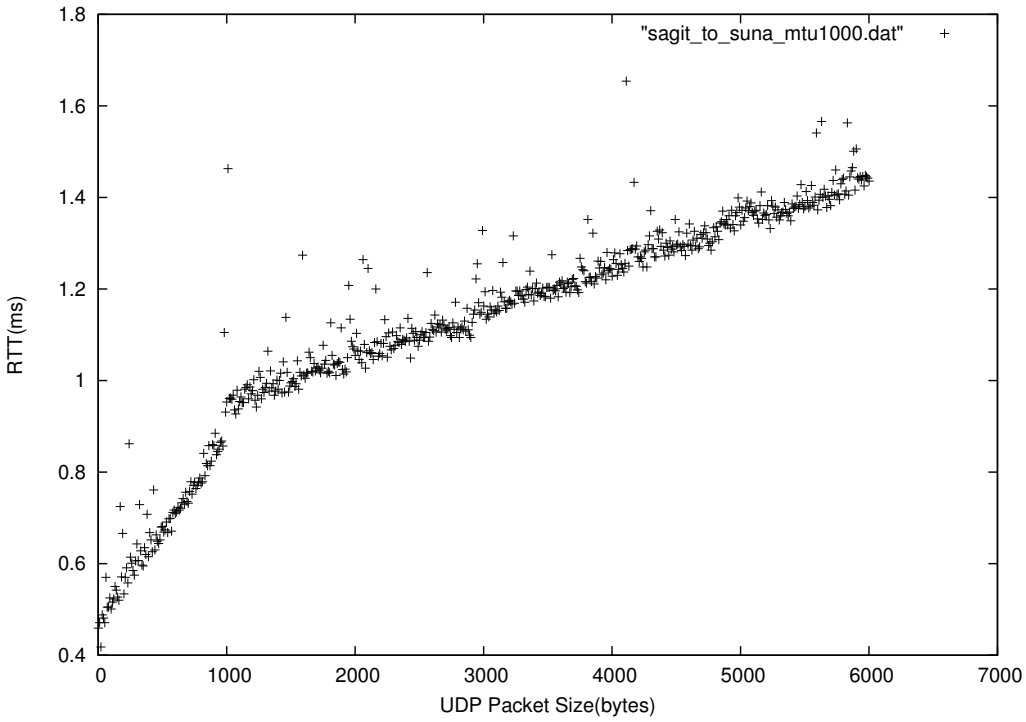


Figure 3.4: RTT from sagit to suna, MTU=1000 bytes

In Fig. 3.5, after the MTU value was set to be 500 bytes, the RTT over packet size threshold value also changed to be 500 bytes.

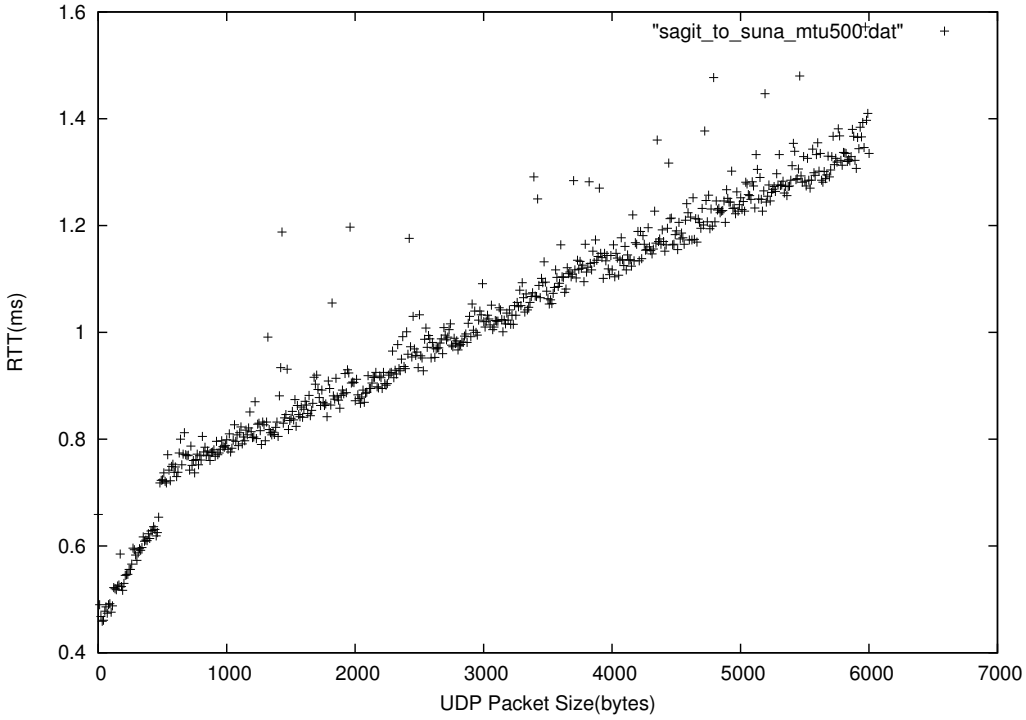


Figure 3.5: RTT from sagit to suna, MTU=500 bytes

To prove that this is not just a unique case for one machine or one network path, another set of measurements were repeated on different pairs of machines and network links. The network paths and machines involved in the further measurements are listed in Table. 3.2.

The graphs of these 6 sample measurements are shown in Fig. 3.6. The result from these samples provides the following observations about the threshold of probing packet size, which affects RTT measurements. Assuming the threshold is called M:

1. The threshold M exists only on the physical network interface. The experiments on loopback interface or other virtual interfaces(e.g. NAT in VMware) did not reveal the effects of M.

Index	Network Link	RTT by *ping*	Description
a	sagit → tokxp	126 ms	NUS campus to APAN Japan[a]
b	sagit → cmui	238 ms	NUS campus to CMU USA [b]
c	sagit → ubin	0.262 ms	local network segment
d	tokxp → jpfreebsd	0.552 ms	APAN Japan to ftp server in Japan
e	helene → atlas	0.196 ms	the same switch
f	sagit → localhost	0.041 ms	test on loopback interface

Table 3.2: Network Paths for RTT Measurements

[a]Asia Pacific Advanced Network, Japan Consortium
[b]Carnegie Mellon University, USA

2. The value of M is very close to the MTU value on the network interface.

3. When the probing packet size $S \leq$ the threshold M, the round trip time has a higher ascending rate. If $S \geq M$, the slope of the RTT over packet size curve will be reduced to a lower value.

4. If the base RTT value is significantly large, in the factor of 10 ms, or the variation of the RTT value is high, the effects of threshold M will be shadowed, which makes M hardly noticeable.

Through these measurements, we believe that the network delay representation in Formula(3.4) must be modified to exhibit this effect. We made the following conjecture. The existence of the *RTT-packet size* threshold could come from the initialization procedure, when the kernel starts to pass the probing data bytes to the physical network interface. The initialization time is determined by the size of first network frame in the data stream and the initialization speed. If we call the initialization speed $Speed_{init}$ and add this new factor into Formula. 3.4, we can get Formula. 3.6, which can explain the change of the RTT slope.

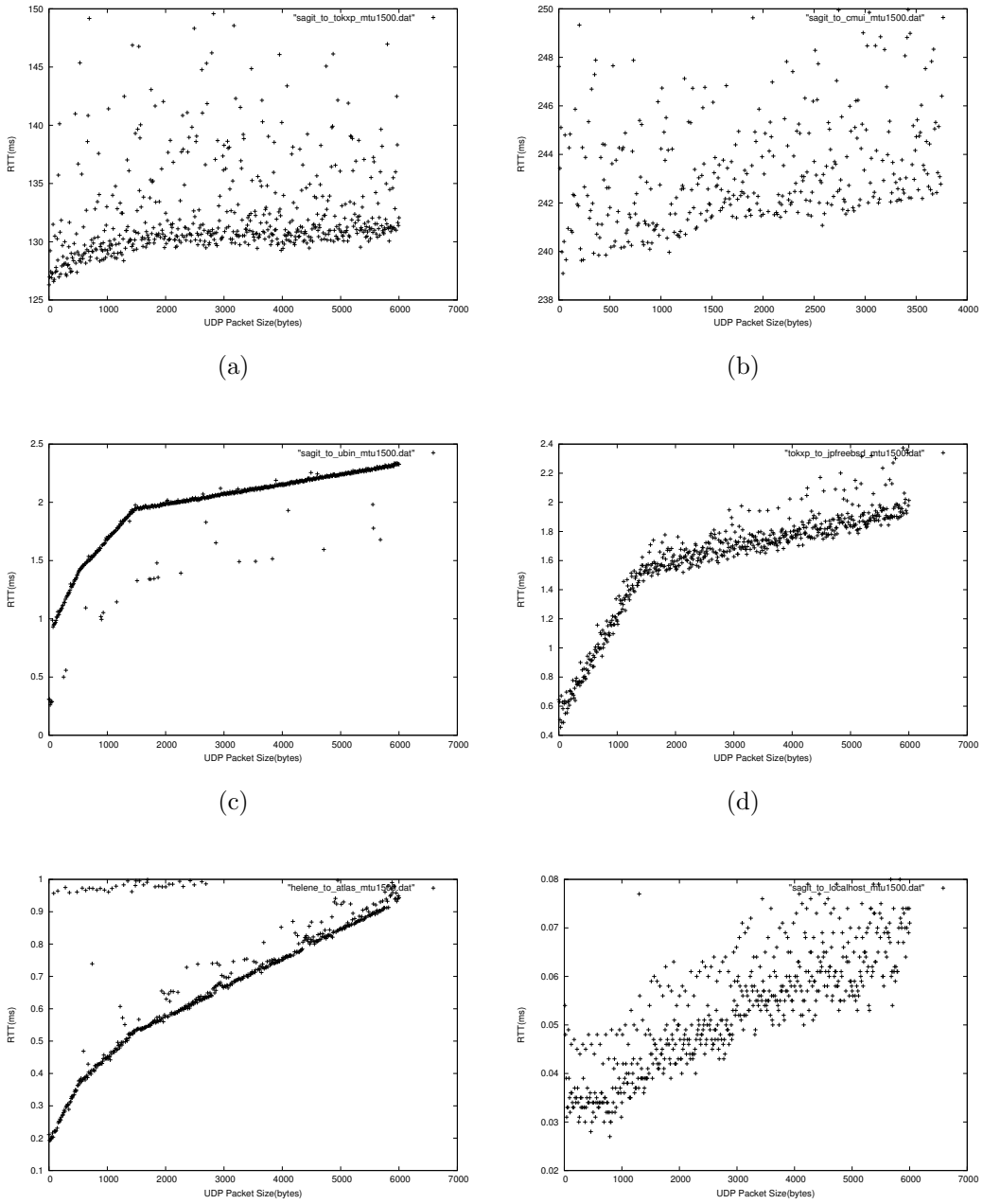


Figure 3.6: RTT Graphs for 6 Sample Network Paths

$$T = \begin{cases} \frac{S}{B} + \frac{S}{Speed_{init}} + Overhead_{sys} + Overhead_{net}, & \text{if S} \leq \text{MTU} \\ \frac{S}{B} + \frac{MTU}{Speed_{init}} + Overhead_{sys} + Overhead_{net}, & \text{if S} > \text{MTU} \end{cases} \tag{3.6}$$

By Formula. 3.6, assuming the slope of RTT during the time when probing packet size $S \leq MTU$ is $Slope_1$, we have $Slope_1 = \frac{1}{B} + \frac{1}{Speed_{init}}$; recall that B is the available bandwidth of the network path. When $S > MTU$, the slope of RTT curve $Slope_2$ is represented by $Slope_2 = \frac{1}{B}$; the initialization time is a constant during that stage. According to Formula(3.5), the slope of the RTT curve will be used as an estimation of the inverse of the available bandwidth $\frac{1}{B}$. In the early stage while packet size $S \leq MTU$, the calculated RTT slope value is not $\frac{1}{B}$ but $\frac{1}{B} + \frac{1}{Speed_{init}}$. As a result, if the probing UDP packet size is less than the MTU value, the available bandwidth we calculated B' will have:

$$\frac{1}{B'} = \frac{1}{B} + \frac{1}{Speed_{init}} \tag{3.7}$$

As we can see from Equation(3.7), $\frac{1}{B'}$ is larger than $\frac{1}{B}$ and $\frac{1}{Speed_{init}}$, given that B is the actual available bandwidth and $Speed_{init}$ is the initialization speed. That will imply $B' < B$ and $B' < Speed_{init}$, which means if the probing UDP packet size $S < MTU$, the estimated bandwidth $B' < B$. In another word, when the probing packet size is smaller than the MTU, the bandwidth measured will be less than the actual bandwidth value under the effects of $Speed_{init}$.

According to this result, the size of the probing packet must be carefully

selected. For the probing packet size S, we propose the following rules:

- The packet size $S > MTU$.
- The sizes of the two UDP probing packets, S_1 and S_2, should be as small as possible. A larger packet size will cause more fragments, which allows more cross traffic packets to intervene the measurements and create confusing results.
- S_1 and S_2 should be selected in the way that the number of fragments generated from these two packets is as close as possible. Although the $Overhead_{sys}$ and $Overhead_{net}$ are considered to be constant in Formula(3.6), the size of the packet may still affect these two factors. This is because packets with different sizes will require different system processing time in the system and routers.

To compare the results under various probing packet sizes, 7 groups of S_1 and S_2 were chosen for experiments as shown in Table. 3.3. The experiment results are presented as a bar chart in Fig. 3.7. From the results, we can see the negative effects from initialization speed $Speed_{init}$, during the time when both S_1 and S_2 are less than MTU value. The bandwidth measured by the first 3 groups is around 20 Mbps, when the actual bandwidth is around 95 Mbps(measured by *pathload*). As $Speed_{init}$ is estimated as 25 Mbps, the first frame from a single UDP packet will be processed at this speed. When both S_1 and S_2 are larger than the MTU value, the measured bandwidth is much closer to the actually available bandwidth, as we can see from the next 4 groups. The 7_{th} group has the best the result, because the probing packet

sizes $S_1 = 1600$ *bytes* and $S_2 = 2900$ *bytes* are the best probing packet size within the set, based on our conclusion above.

Packet Size(Bytes)	Min Bw(Mbps)	Max Bw	Avg Bw
100∼500	18.68	21.10	20.01
500∼1000	17.45	19.71	18.39
100∼1000	17.88	18.79	18.33
2000∼4000	85.77	91.81	88.12
4000∼6000	78.28	90.72	85.18
2000∼6000	82.26	85.21	83.54
1600∼2900[a]	86.49	99.03	92.86
pipechar	95.346		
pathload	96.1∼101.3		

Table 3.3: Bandwidth Measurements using various Packet Size

[a]Optimal Packet Size under $MTU = 1500$ *bytes*

3.3.3 Network Monitor Procedure

In a large computing environment involving many server groups, each server group has an individual network monitor. The network monitor collects the network status information from the network paths linking local servers to remote servers. The operation diagram is given in Fig. 3.8.

Each network monitor is informed about the neighboring network monitors around and probes one another for the delay and bandwidth values along the network paths. The network status record formatted as a table shows the (delay, bandwidth) pairs to each neighboring network monitor. This table contains the network status for network paths from local server group to all the other groups. This information can be utilized by those applications in which the network delay or bandwidth is one of the major concerns.

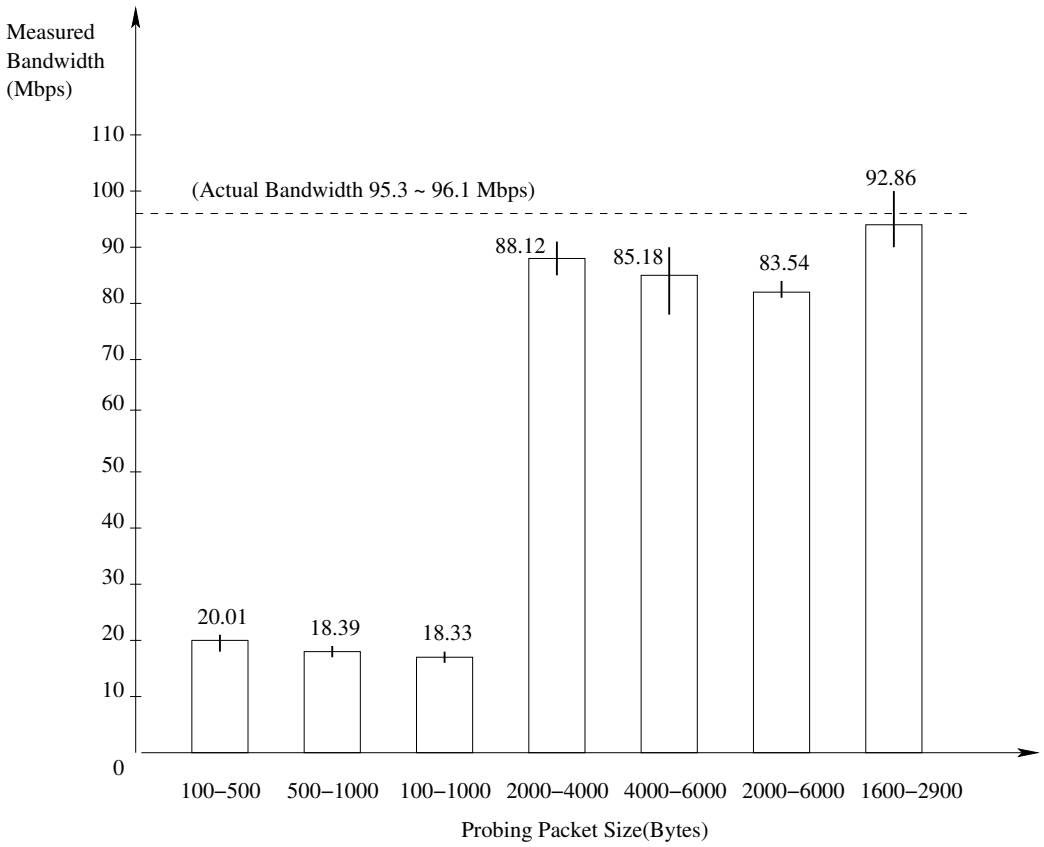


Figure 3.7: Bandwidth Measurements using various Packet Size

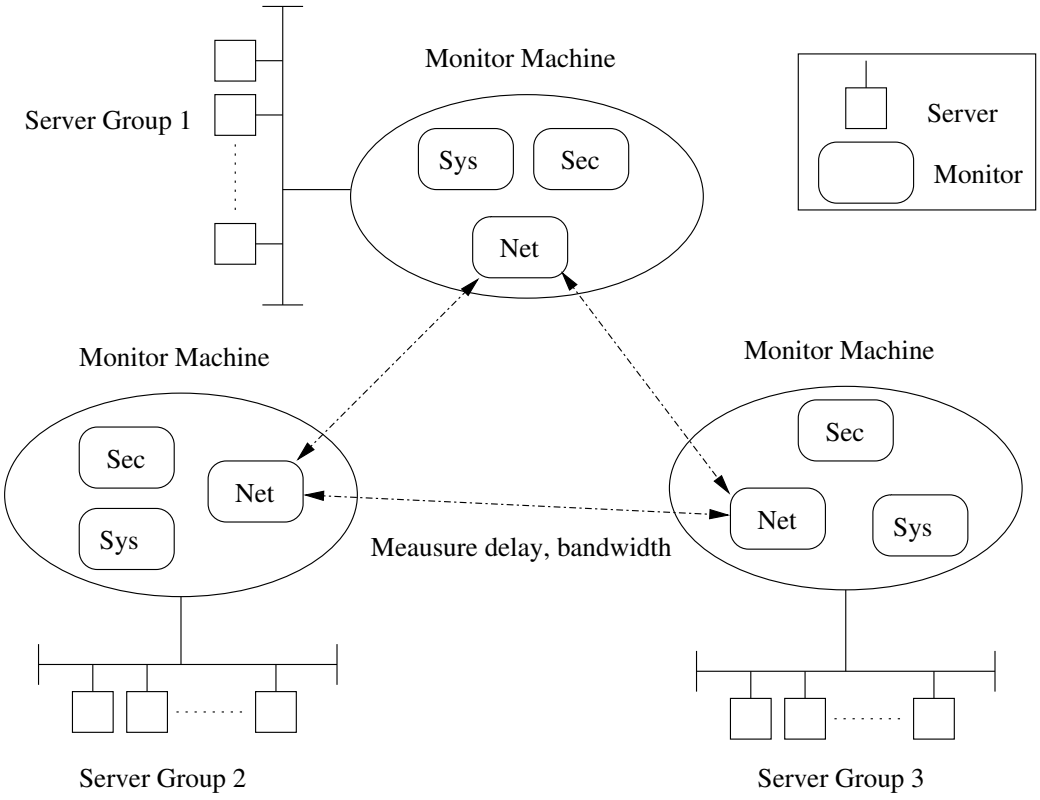


Figure 3.8: Operations of Network Monitor

The network status records for the sample structure in Fig. 3.8 is listed in Table. 3.4.

Net Monitor	netmon-1	netmon-2	netmon-3
Net Status	mon2(delay, bw)	mon1(delay, bw)	mon1(delay, bw)
	mon3(delay, bw)	mon3(delay, bw)	mon2(delay, bw)

Table 3.4: Sample Network Monitor Records

The assumption is that in the local area network, the bandwidth and delay is sufficient for most applications. Only in larger area networks, where multiple servers from various locations are joining to work for the same task, the network condition will influence the performance. In those cases, users may specify a request of "($delay < 20ms$) and ($bandwidth > 10Mbps$)" to avoid sub-optimal servers. The user request handler - *wizard* will look at the statistics collected by network monitors to check the availability of those qualified servers. Traditional server selection techniques normally do the round-robin blindly, or count the number requests/connections handled by each server, ignoring the user's requirement. From the user's perspective, the algorithms utilizing network status records can provide better response time and higher throughput, which is a major improvement from the traditional server selection techniques. Classic server selection techniques normally do the round-robin blindly, or count how many requests have been handled by the server, how many connections the server has made, and ignore the user's requirement.

The probing setting must be carefully configured by the administrator. The probing interval should be determined by the number of the server groups. More server groups in the computing environment will create more

network paths to probe. The total number of probes is $P_n^2 = n \times (n-1)$, given that n is the number of server groups. The probing interval should get larger as the number of network paths increases. The network probing procedure should be done in a sequential order. Multiple probes should not run simultaneously. Or else it will introduce high extra network traffic and cause interference between concurrent probes.

3.4 Security Monitor

3.4.1 General Security Issues

The security issue is not one of the main concerns in Smart TCP socket library. The network access control, operating system patching and the application hot-fix should be handled by other system components. In the current implementation of the Smart TCP socket library, the security monitor reads the security records from an dummy security log. The log file contains the server names and the correspondingly security levels, which is an integer representing the clearance level of each server. We leave a framework of the security component to be open, such that third party security components can be plugged in with minimal modification.

The security information will be reported from those security "probes". Cisco has presented a Network Admission Control(NAC) mechanism[cisco04] to protect the servers in a network from being attacked by worms, computer viruses and various hacker attacks. The Cisco Security Agent will be installed to the network servers to collect the operating system version, patch level and

hot-fix information. Other software clients such as anti-virus software can be integrated with Cisco Security Agent to provide further information about viruses or worms found in the servers. These reports will be sent to a Trust Agent for further action. If the security reports collected by the security agents and anti-virus software can be sent to the security monitors in our Smart TCP socket library, users can also create precise requests on server security.

3.4.2 Security Techniques

Currently there are two conventional methods to collect system/network security information. One is *nmap*(Network Mapper) based probings for network scanning; the other method is registry scanning to diagnose local machines. In *nmap* based probing, the probing software sends out packets and analyzes the response from the target host and compares the server's response with the local fingerprint database. This fingerprint database stores the typical responses that most main stream operating systems may generate. The classic probing measures, including TCP FIN probe, TCP ISN Sampling, ICMP Message quoting, are fully explained in Fyodor's paper[fyodor98]. Together with Port Scanning techniques the various services running on the servers can be checked for any security holes. A sample output from *nmap* program is listed below:

```
Starting nmap V. 2.54BETA31 ( www.insecure.org/nmap/ )
Interesting ports on debian (127.0.0.1):
(The 1550 ports scanned but not shown below are in state: closed)
```

```
Port       State       Service
9/tcp      open        discard
13/tcp     open        daytime
22/tcp     open        ssh
37/tcp     open        time

Remote operating system guess: Linux Kernel 2.4.0 - 2.4.17 (X86)
Uptime 2.245 days (since Sat Jun 26 10:41:28 2004)

Nmap run completed -- 1 IP address (1 host up) scanned in 2 seconds
```

The registry can be scanned to collect local security information, a technique commonly used in Windows systems. One example is the Network Security Scanner(NSS) from GFI[gfi04]. The scanner checks the registry to extract security report about OS version, patch list, service ports opened and possible vulnerabilities in the local machine. Compared with the network probing method, the registry scanning method is more time efficient and accurate. However, it supports only Windows based systems.

Apart from the possibility to integrate the security agents into new socket library, the task of controlling the network access, managing network services, detecting the service bugs and installing hot-fixes should be handled by a separate group of programs.

3.5 Transmitter and Receiver

The transmitter and receiver work together to transfer the information from the monitor machines to the wizard machine. The operations of these two components are demonstrated in Fig. 3.9.

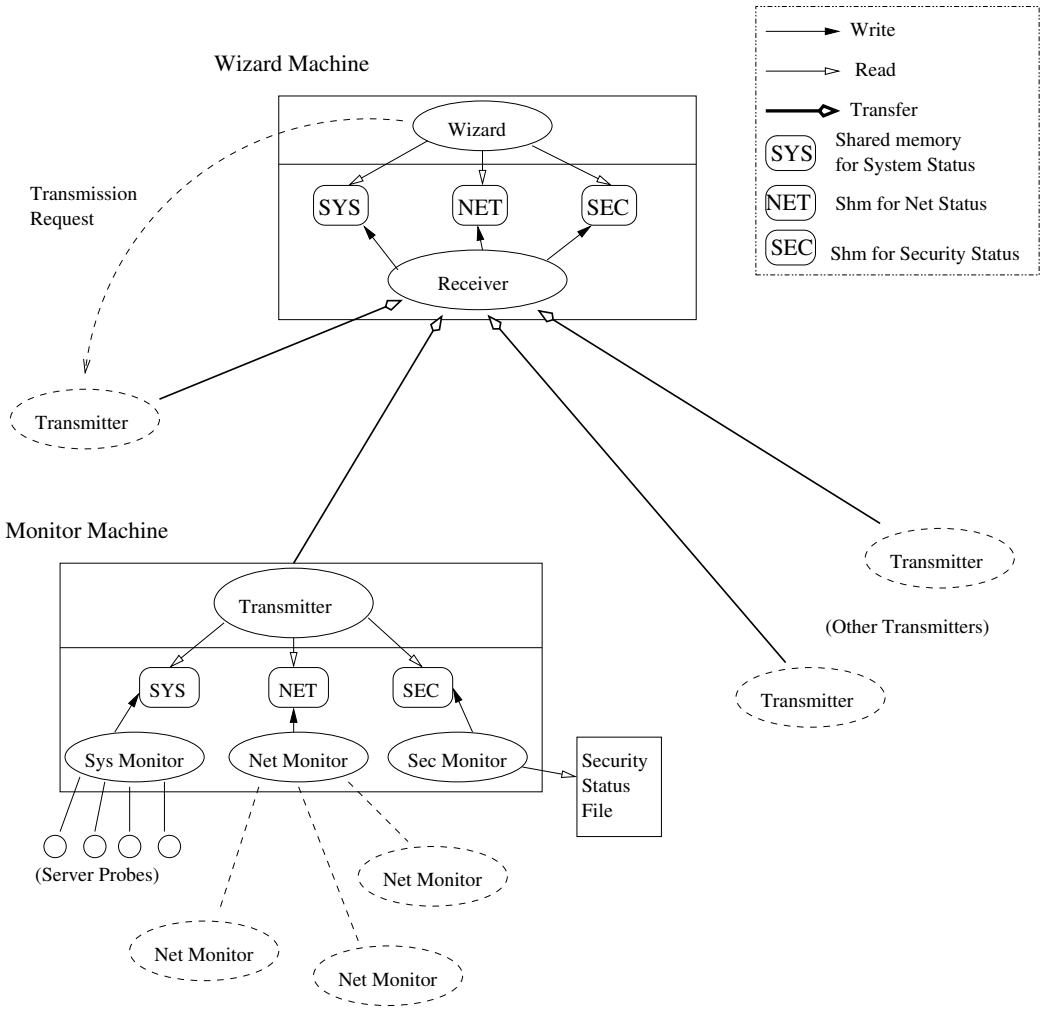


Figure 3.9: Interactions between the Transmitter and Receiver

3.5.1 Transmitter

The transmitters are running on the monitor machines where the 3 monitors: system, network, security monitors reside. The 3 monitors write the 3 types of status records into the shared memory regions. The transmitter reads the contents of those 3 memory regions and transfers the data to the receiver running on the wizard machine.

The records are copied out from the memory space and sent in binary format. The character string is not used to represent the data, as each monitor may handle a large number of servers. The binary to ASCII conversion is resource consuming and less efficient. This binary transmission scheme requires that the two machines with the transmitter and receiver running must have the same hardware architecture in order to avoid the Endian issues. For instance, the number $0xAABB$ in big endian machines will become $0xBBAA$ in little endian machines. The data type units should also be consistent in the two machines. A 64-bit long integer in machine A may result in a value overflow in machine B, in which a long integer has only 32 bits.

TCP protocol is used to transmit the server status and network status information *transmitters* to *receivers*. The format for data transmission is `[type, size, data]`. *Type* and *size* fields are transmitted first, so the *receiver* can determine the amount of memory that should be allocated to store the *data* field. Since the *data* field is in binary format, the contents can be directly copied to shared memory space in the *receiver*.

The transmitter has different behaviors under the centralized and the distributed mode. In the centralized mode, the transmitter actively sends

data reports from system, network, security monitors to the receiver at a regular interval. In the distributed mode, the transmitter will listen in passive mode, waiting for the transmission request from the wizard. The reports are sent back only when a transmission request from the wizard is received. The idea is that in the centralized mode, when the servers are located in a small area, we can instantly get the status updated and improve the request processing time. In the distributed mode, the server groups are located sparsely in a large area. The user request will come less frequently, so regular transmission of the large amount of status records may cause unnecessary network load. The two operating modes of transmitters and receivers make them adaptable to different situations.

3.5.2 Receiver

The receiver listens on the service port to wait for incoming reports from the transmitter. According to the contents of the incoming data stream, the receiver creates the corresponding data structures to store the information and updates the data structures in the shared memory. In this way, the receiver can maintain the identical shared memory contents as what is in the transmitter. The wizard can directly use the contents as if they were generated locally.

In the centralized mode, there is one receiver running together with the wizard in the same machine. The receiver periodically obtains the status reports from the transmitters and refreshes the shared memory accordingly. In the distributed mode, there could be multiple receivers and wizards. A

wizard triggers all transmitters participating in the computing task to send updated reports to the receiver, upon the incoming of a new user request.

3.6 Wizard and Client Library

The *wizard* program will be used to handle the server requirement from the client library directly. These two components will be described in this section.

3.6.1 Procedures of Wizard

The *wizard* program, running as a daemon, waits for the user request at the service port and processes the user requests sequentially. The underlying protocol used is UDP protocol due to the low overhead. Also when the incoming user requests become enormous, the TCP based server will have quite a few "TIME_WAIT" connections left. "Too many files opened" error may occur during peak time to prevent new connections from being established.

The main procedure of the *wizard* contains the following steps:

1. The *wizard* listens on the service port for the user's request. The format of user request is shown in Table. 3.5

Sequence Num	Server Num	Option	Request Detail

Table 3.5: Format of User Request

Sequence Num is the random number generated by the client library to identify the current user's request. In the reply message for that particular request, the same sequence number will be used. This can ensure that when multiple user requests are issued from a single client

machine, the client library can make a correct match between requests and replies.

Server Num is the number of servers required. The *wizard* will try to find the exact number of available servers as candidates. There is an upper bound for this number, because the server list is sent back in the UDP message, which is not reliable when the message becomes long. Currently the limit is set to be 60.

Option field is used to provide additional user options in special situations, like when the number of returned servers is less than requested or when the user wants to use some predefined server requirement templates. *Request Detail* contains the detailed user request in character string format. It is the full description about what kind of servers are wanted.

2. The wizard reads the shared memory contents to get the data structures updated. In the *centralized* mode, the shared memory area is updated by the *receiver* program periodically. In the *distributed* mode, the *wizard* has to issue an update request to the transmitters of all server groups for updates. There are 3 data structures in *wizard*: *sysdb* for system status of the servers, *netdb* for network metrics of the monitors and *secdb* for the security levels of the servers. Fig. 3.10 illustrates the format of the 3 structures.

3. The server status will be loaded and compared with the user's requirement. The processing of user's requirement has two steps: lexical analysis and syntactical analysis[lexyacc92]. The lexical analysis will parse

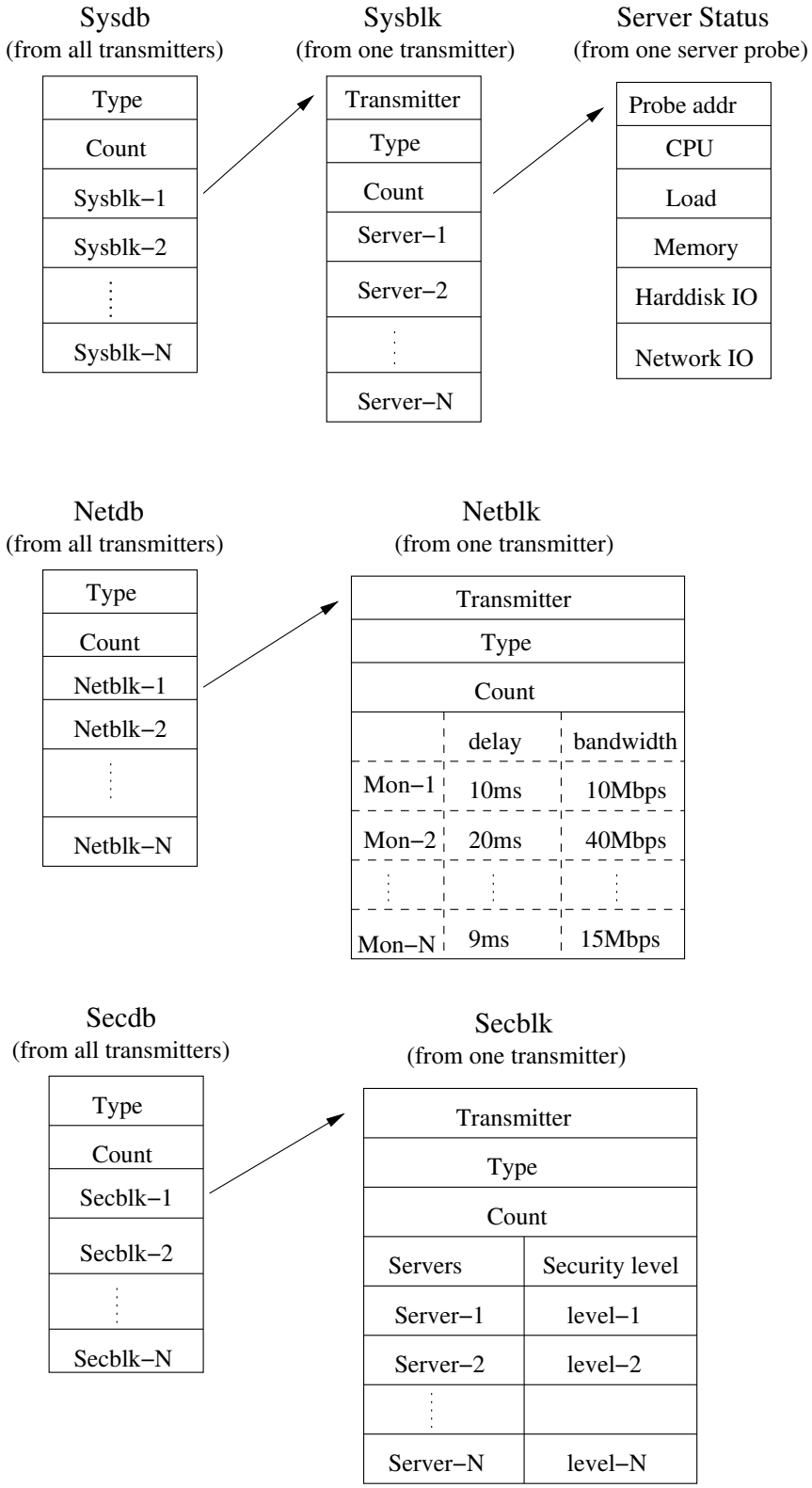


Figure 3.10: Format of Status Record Structures

the user request contents into small units named *tokens*.

- "`#.*`" - any strings after a "`#`" sign are considered as comments, ignored by the parser. Users may write the comments to group the statements in proper order.
- " " and "`\t`" are white space characters, which are ignored.
- "`[0-9]+|[0-9]+\.[0-9]`" is classified as numerical type.
- "`[a-zA-Z]+[a-zA-Z_0-9]*`" is be parsed as a variable. There are 3 types of variables: temp variables, user-side variables and server-side variables. Temp variables are defined at the requirement context, used to assist the description of the requirement details. Server-side variables are predefined at the *wizard* side, whose value will be given by the status reports from the monitors. The third type is the user-side variables, whose values are assigned by the user. Currently, there are two groups of user-side variables: trusted-servers and untrusted servers. The trusted servers will always be selected first when available and the untrusted servers will be avoided by the *wizard*.
- "`<num>.<num>.<num>.<num>`" and "`<string>.<string>...<string>`" are considered as *network address*, which represents the address of the servers in user's preference list. The actual validation of the network address will be done by the socket function call in system.
- "`>, >=, <, <=, ==, !=, &&, ||`" are parsed as the logical operator. The return value for logical operation can be either *True*

or *False*.

After the lexical analysis step is done, the syntax of the user requirement will be checked. The basic rules for the statements are given as below:

- Users can use variables to do mathematical calculations. The value of the mathematical operations will be the numerical. In this case, the normal operations as $+$, $-$, $\times$, $\div$ will remain the same.
- A statement can be either a logical statement or non-logical. The return value of a logical statement will be used to decide whether a server is qualified. A statement is logical, only if the main operator is a logical operator. For example `"(a+b)<=b"` is a logical statement but `"a+(b<c)"` is not a logical statement.
- The simple assignment statement can be used to define temp variables. If an uninitialized temp variable is used in the logical statement, the whole statement will be considered as a false statement.
- If multiple lines of statements are specified, the server is qualified, only when all the logical statements return the *true* value.

4. Once the available servers are selected, a server list will be built and sent back to the client. The format of the reply message is given in Table. 3.6.

Sequence Num is the sequence number of the reply message, which is identical to the one in the original request message. *Server Num* will be the

Sequence Num	Server Num	Server-1	...	Server-n

Table 3.6: Format of Reply Message from Wizard

number of servers returned. Following that, it is the list of candidate server addresses.

3.6.2 Functions of Client Library

The client library interacts with the user directly. The user provides the server specifications and the client library will find the optimal servers with the help from the *wizard*. The main procedure of the client library contains the following steps:

1. The client library reads the user's requirement from the requirement file. A sample user requirement may look like the following:

```
host_system_load1 < 1
host_memory_used <= 250*1024*1024
host_cpu_free >= 0.9
#ldjfaldjfalsjff  #akldjfaldfj
#some comments
host_network_tbytesps < 1024*1024       # for network IO
# comments
user_denied_host1 = 137.132.90.182
user_preferred_host1 = sagit.ddns.comp.nus.edu.sg
#
```

   This user requirement contains 4 server-side variables: *host_system_load1*, *host_memory_used*, *host_cpu_free*, *host_network_tbytesps* and 2 user-side

variables: *user_denied_host1*, *user_preferred_host1*. There are in total 22 server-side variables and 10 user-side variables available. Together with the built-in functions such as *exp*, *sin*, *cos* and log_{10}, users can write very sophisticated expressions about the servers. In the example above, the user requires that the server's system load in the last 1 minute should be less than 1, memory space used in the server should be below 250 *Mbytes*, the free CPU time should be at least 0.9. Also the user will deny the selection of the server with IP address "137.132.90.182" and the preferred server is "`sagit.ddns.comp.nus.edu.sg`".

2. The client library then attaches the random sequence number, server number plus the option supplied by the user to the requirement details. The user request message as described in Table. 3.5 will be sent as a UDP message to the *wizard*.

3. Once the request message is sent, the client library will wait for the reply from the *wizard* in the format as given in Table. 3.6. The sequence number will be compared with the original one, in order to ensure that this is the expected reply message. The server number in the reply message will also be checked. If the returned server number is equal to the one in the request message, it means all the required number of servers have been found. If the number is less, client library will take different actions based on the option from the user.

4. The client library will try to make a connection to the service port of each server in the candidate list. The connected sockets will be returned to the original caller in the user's program. The user's program and

the actual service program running on the servers should be aware of how to interact through the list of connected sockets. Hence, it is the user's responsibility to tell what kind of servers are optimal.

Chapter 4

Implementation Issues

A few key implementation issues of the library components will be discussed in this chapter.

4.1 Server Probes

The 5 `/proc` file system nodes revealing the system configuration and workload status are listed below:

```
loadavg_fname = "/proc/loadavg"
cpuusage_fname = "/proc/stat"
memusage_fname = "/proc/meminfo"
diskio_fname = "/proc/stat"
netio_fname = "/proc/net/dev"
```

The `/proc/loadavg` file gives the average system load of last 1, 5 and 15 minutes which indicates the average workload. The proc entry `/proc/stat`

gives us the CPU time usage of the user process, system process, and the idling processes. Compared with the CPU usage figures, the load average values are considered as a better estimation measure for system workload, because they describe how busy the server is in a long term instead of at a particular moment. However, the CPU time usage figures can reflect any change in CPU usage instantly. So the combination of these two sets of parameters will be critical for CPU intensive tasks.

The memory usage information is provided in `/proc/meminfo`. It shows the usage pattern of the physical memory. This information will be necessary for those memory intensive computation tasks, such as *SuperPI*[superpi04]. A sample memory status comparison before and after running *SuperPI* is given in Table. 4.1. `Mem1` is the status before we started *SuperPI* and `Mem2` shows how much more memory was acquired by the program. Depending on the algorithms, memory intensive applications will experience poor performance in the memory bound servers.

	total	used	free	shared	buffers	cached
Mem1	262213632	121085952	141127680	0	18284544	82911232
Mem2	262213632	258310144	3903488	0	745472	231075840

Table 4.1: Memory Usage before and after SuperPI

The information we collect from `/proc/net/dev` and `/proc/stat` is required for data intensive applications. The `disk_io` entry in `/proc/stat` shows the total number of read and write requests, and the number of disk blocks accessed by read and write requests. Monitoring these figures can help us to identify the current hard disk activities. The entry `/proc/net/dev` lists all the network interfaces available for the current server and amount of data

going through each interface. For data intensive applications, both the hard disk activity and the network bandwidth usage will be important.

The server probes scan through these 5 `/proc` files at a regular interval of 10 seconds and send the server status reports back to $monitor_{sys}$ - the system status monitor. A server failure is detected, if any probe fails to report after 3 consecutive intervals.

4.2 Monitors and Wizard

The 3 monitors $monitor_{sys}$, $monitor_{net}$, $monitor_{sec}$ and the *transmitter* are running in the monitor machine. The *receiver* and the *wizard* are running on the wizard machine. This means altogether there would be 6 port numbers to use. Table. 4.2 gives the current assignment of these port numbers.

Machine	Monitor Machine				Wizard Machine	
Component	Mon_{sys}	Mon_{net}	Mon_{sec}	$Trans_{pass}$	Receiver	Wizard
Port	1111	1112	1113	1110	1121	1120

Table 4.2: Ports used by Monitors and Wizard

The 3 monitors write the records into the shared memory space and the transmitter transfers the data from the monitor machine to the receiver on the wizard machine. The receiver then writes back the data received into another set of shared memory spaces in the wizard machine for the *wizard* to access. To enable concurrent read and update of the shared memory contents, semaphores are used to lock and unlock the related resources. The keys we assign for both semaphores and shared memories are the same for one type of records. The shared memory keys and semaphore keys allocated

are shown in Table. 4.3.

Location	Monitor Machine			Wizard Machine		
Type	System	Network	Security	System	Network	Security
Key	1234	1235	1236	4321	5321	6321

Table 4.3: Keys for Semaphores and Shared Memory Spaces

According to this key assignment scheme, there would be no conflict on the system resources, even if we run all the monitors, transmitter, receiver, and the wizard program in the same machine.

4.3 Server Requirement Parser

The *server requirement* for a particular application represents the qualification rules for the wizard to decide which servers should be selected. The requirement handling procedure contains two steps: lexical parsing and semantic analysis. The parser is implemented by using flex[gnuflex00] and bison[gnubison03] provided by GNU project[gnuproject04].

The server requirement is first parsed into small tokens. We have the following token types: *comment sign*, *white space*, *NUMBER*, *NETADDR*, *UNDEF*, *VAR* plus the logic operators as defined in the C programming language. *NETADDR* is created for users to write IP addresses in the numerical format or domain names in the string format. *NUMBER* is used to represent values of the server status attributes. The variables defined in the parser will be in *VAR* type, whose value has been given or will be provided by the server reports or users. The *UNDEF* variables are those undefined variables whose values are not given anywhere. Users should pick up the correct server

attributes to construct a proper requirement. The *newline* symbol '\n' is used to signal the end of a statement. The basic rules for token parsing are given in Fig. 4.1.

```
#.*     { ; /* ignore comments */ }
[ \t]       { ; /* ignore white spaces */ }

[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+ |
[a-zA-Z]+[a-zA-Z_0-9]*\.[\.a-zA-Z_0-9]* { return NETADDR; }
[0-9]+ |
[0-9]+\.[0-9]+    { return NUMBER; }
[a-zA-Z]+[a-zA-Z_0-9]*  { return UNDEF or VAR; }

\&\&  { return AND; }
\|\|  { return OR; }
\>   { return GT; }
\>\=  { return GE; }
\=\=  { return EQ; }
\!\=  { return NE; }
\<   { return ST; }
\<\=  { return SE; }

\n    { return '\n'; }
.    { return yytext[0]; }
```

Figure 4.1: Lexical Rules for Parsing Tokens

For the semantic analysis, the detailed semantic rules are shown in Fig. 4.2. These *yacc* rules are built based on the example in [brian84]. In the server requirements, each line is considered as a statement, which can be *logical* or *non-logical*. Logical statements are the ones whose last operator is a logical operator, for example $\neq$, $\geq$ and $<$. These logical statements return boolean values, *true* represented by numerical value of 1, and *false* represented by 0. The logical statements are mainly used to compare the values of server

attributes with the user's specification. The parameters appearing in these statements come from either the user side or the server side. The server side parameters are the server attributes defined in the server reports, which is extracted through the system interface. The user side parameters are those provided for users to write additional requirements. At this moment, we have two sets of user side parameters: *preferred_hosts* and *rejected_hosts*. Users can tell the wizard which are the servers that should be used first and which are the ones in blacklist that should be avoided.

In non-logical statements, users can write intermediate steps, such as defining temporary variables and doing calculations. The return values for non-logical statements will not be used to determine if the server is qualified. Yet the variables that have been modified in these statements may affect the final decision. Parenthesis are also provided for controlling the operation precedence. Basically, logical statements contain logical comparisons and non-logical statements contain mathematical operations like $+$, $-$, $\times$, $\div$ and assignment statements.

Finally, the return values of all the logical statements are examined. The principle is that only if all the user requirements are fulfilled, the server can be taken as a candidate. However, the user should be responsible for the statements they give to the *wizard* program. A meaningless statement like `100 > 0` will make any server as a qualified candidate. And it would not be a good practice if we pick servers with `CPU usage > 99%` for CPU intensive tasks. Users must be very clear about the the algorithms in the applications, settings of the programming environment and the acceptable performance result. An automatic analysis for the complexity of the program

code would be another challenge and is still under active research[alkindi00]. It is necessary for inexperienced users to perform a few experiments first, in order to figure out under what condition the satisfying results can be achieved.

```
list:   /* nothing */
    | list '\n'
    | list expr '\n'   {  printf("\t%lf\n", $2);
                          if(logic == 1)
                          { server_ok *= $2; logic = 0; }
                       }
    | list error '\n'   { yyerrok; }
    ;

asgn: VAR '=' expr { $$ = $1->u.val = $3; $1->type = VAR; logic = 0; }
    | UPARAM '=' expr { $$ = $1->u.val = $3; logic = 0;
                        store_uparams($1->name, $1->u.val); }
    ;

expr: NUMBER { logic = 0; }
    | NETADDR { logic = 0; }
    | UPARAM { $$ = $1->u.val; logic = 0; }
    | PARAM { $$ = $1->u.val; logic = 0; }
    | expr AND expr { $$ = ($1 && $3); logic = 1; }
    | expr OR expr { $$ = ($1 || $3); logic = 1; }
    | expr EQ expr { $$ = ($1 == $3); logic = 1; }
    | expr NE expr { $$ = ($1 != $3); logic = 1; }
    | expr ST expr { $$ = ($1 < $3); logic = 1; }
    | expr SE expr { $$ = (($1 < $3) || ($1 == $3)); logic = 1; }
    | expr GT expr { $$ = ($1 > $3); logic = 1; }
    | expr GE expr { $$ = (($1 > $3) || ($1 == $3)); logic = 1; }
    | VAR { if($1->type == UNDEF)
                execerror("undefined variable", $1->name);
            $$ = $1->u.val; logic = 0; }
    | asgn
    | BLTIN '(' expr ')'    { $$ = (*($1->u.ptr))($3); logic = 0; }
    | expr '+' expr { $$ = $1 + $3; logic = 0; }
    | expr '-' expr { $$ = $1 - $3; logic = 0; }
    | expr '*' expr { $$ = $1 * $3; logic = 0; }
    | expr '/' expr {
            if($3 == 0.0) { execerror("division by 0", ""); logic = 0; }
            $$ = $1 / $3; logic = 0; }
    | expr '^' expr { $$ = Pow($1, $3); logic = 0; }
    | '(' expr ')' { $$ = $2; /* this op will not change logic value */ }
    | '-' expr %prec UNARYMINUS { $$ = -$2; logic = 0; }
    ;
```

Figure 4.2: Semantic Rules for Parser

Chapter 5

Performance Evaluation

In this chapter, we will present the experiment related issues, including the testbed configuration, the machines settings and the experimental results.

5.1 Testbed Configuration

5.1.1 Networks

The 11 machines in the testbed are located in 6 different network segments. The private network segments `192.168.1.0/24` to `192.168.5.0/24` are located in the Communication and Internet Research lab at NUS. The remote host *sagit* is in the School of Computing network `137.132.81.0/24`, connecting to the testbed through the gateway *Dalmatian*. All networks are 100 Mbps Ethernet. The complete network topology of the testbed is given in Fig. 5.1.

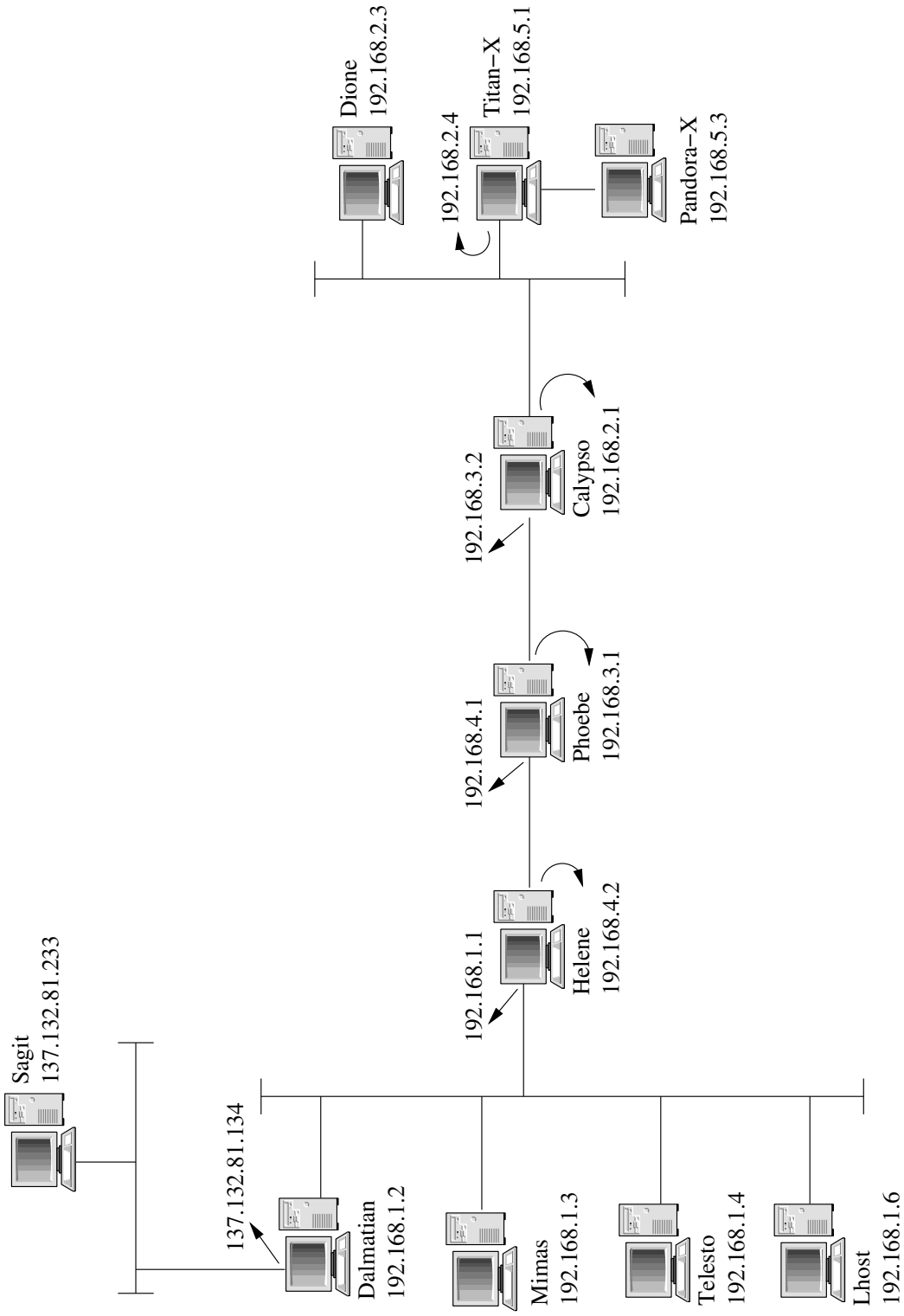


Figure 5.1: Network Topology of the Testbed

5.1.2 Machines

There are in total 11 machines in the testbed running on the Linux operating system. The hardware configurations of the Linux machines used in the experiments are listed in Table. 5.1.

Hostname	CPU/bogomips	RAM	OS
Sagit	P3 866MHz/1730.15	128MB	Debian Linux 3.0r2
Dalmatian	P4 2.4GHz/4771.02	512MB	Redhat Linux 8.0
Mimas	P4 1.7GHz/3394.76	192MB	Redhat Linux 9.0
Telesto	P4 1.6GHz/3185.04	128MB	Redhat Linux 7.3
Lhost	P3 866MHz/1730.15	128MB	Redhat Linux 9.0
Helene	P4 1.7GHz/3394.76	256MB	Redhat Linux 9.0
Phoebe	P4 1.7GHz/3394.76	256MB	Redhat Linux 9.0
Calypso	P4 1.7GHz/3394.76	256MB	Redhat Linux 9.0
Dione	P4 2.4GHz/4771.02	512MB	Redhat Linux 7.3
Titan-X	P4 1.7GHz/3394.76	256MB	Redhat Linux 7.3
Pandora-X	P4 1.8GHz/3591.37	256MB	Redhat Linux 9.0

Table 5.1: Configuration of the Testbed Machines

5.2 System Resource Required

To measure the system resource required by each of the library components, a set of sample tests were done on the *Dalmatian* host. The system load, CPU usage and memory usage were monitored through `top` command in Linux. The network bandwidth usage was measured by *traffic dumper*, a network traffic monitor developed using *libpcap*[libpcap04]. The detailed system resource usage figures are listed in Table. 5.2.

For system probes, the probing interval is set to be 2 seconds. The size of the probing message is around 190 bytes. The network bandwidth usage for

Program	CPU	Memory	Net bandwidth
System Probe	$< 0.1\%$	8 KB	$0.5 \sim 0.6$ KBps(UDP)
System Monitor	0.7%	8 KB	5.7 KBps(UDP)
Network Monitor	$< 0.1\%$	8 KB	5.6 KBps(UDP)
Security Monitor	$< 0.1\%$	8 KB	(not used)
Transmitter	$< 0.1\%$	8 KB	1.2 KBps(TCP)
Receiver	$< 0.1\%$	92 KB	1.2 KBps(TCP)
Wizard	0.1%	96 KB	$<$ 1 KBps(UDP)

Table 5.2: System Resource used with 11 Probes Running

each system probe program is around $0.5 \sim 0.6$ KBps, which could be reduced by increasing the probing interval. The system monitor collects the probing messages sent by the probes. As a result, the number of server reports to a particular system monitor will directly affect the resources consumed in a monitor machine. With 11 probes reporting, the system monitor program uses 0.7% of the CPU time. Each probe message will be parsed into a server status structure, which is 204 bytes long. The total memory usage will be determined by basic memory used by monitor itself and number of probing reports received. The network bandwidth used by the system monitor is equal to the sum of the bandwidth used by all the probes.

For the network monitor, the network metrics are measured by sending probing packets periodically. The CPU and memory usage is negligible, while the network bandwidth usage is determined by the size of the probing packets and the probing frequency. The current probing packet size is 1600 and 2900 bytes and one probe is done after every two seconds. The bandwidth acquired is 2.8 KBps. For the transmitter, the CPU and memory usage is insignificant. The receiver program requires much more memory space, because it maintains the status reports and updates the contents in

local shared memory. To transmit 11 server status reports, 1 network status report and 2 security reports, the total bandwidth usage is 1.2 KBps, at a transmission interval of 2 seconds. The *wizard* program consumes more CPU time and memory than any other component, as it maintains all data structures dynamically and processes the incoming user requests iteratively. The average CPU usage is 0.1% and memory usage in the sample test is 96 KB. The network bandwidth acquired is related to the length of the request and reply messages, determined by the list of selected servers, the number of clients using the wizard, and the frequency of the incoming user requests. In the sample run, given that the user request is 150 bytes long and the return message contains 11 host entries, the measured bandwidth usage is less than 1 KBps.

5.3 Experiment Results

Two sample programs have been developed to verify the effectiveness of the Smart TCP library – a matrix multiplication program and a massive downloading program which makes use of parallel TCP connections to multiple servers.

5.3.1 Matrix Multiplication

A square matrix multiplication program was developed to conduct the experiments of the Smart TCP library. It contains a local computation mode and a distributed computation mode. For the local mode, the 2 input matrices will be multiplied in a vector multiplication way. For the distributed mode,

the entries in the input matrices are transferred to the available servers for computation. The result entries will be sent back and stored for output. The implementation of the matrix program is provided in Appendix. C.1.

The execution time of the matrix multiplication reflects the performance. It is determined by the CPU power, amount of memory available, the algorithm applied in the program and programming environment used, especially the compiler type and the optimization option enabled.

Since all the 11 machines used in the matrix multiplication tests have different hardware configuration, the benchmarking step is conducted to measure the computational power for each of them. The benchmark matrix size is 1500×1500 and block size 200×200. The full details are displayed in Fig. 5.2. The chart shows that for our matrix multiplication program, the P3 866MHz and P4 2.4GHz CPUs have better performance than the P4 1.6GHz $\sim$ 1.8GHz ones. This benchmark result can be used to analyze the experimental results.

Four sets of matrix multiplication experiments were conducted for comparison of the execution time without and with the assistance of our socket library, including 2 vs 2, 4 vs 4 and 6 vs 6 under zero workload and 4 vs 4 under non-zero workload.

1. **2 vs 2 without Workload**. In this experiment, 2 servers were selected to compute a matrix with the dimension 1500×1500 by using 600×600 blocks. With the help of the Smart socket library, the user can ask for the servers with fastest processors for this CPU intensive task. The server requirement contains `bogomips > 4000`, `cpu_free > 0.9` and

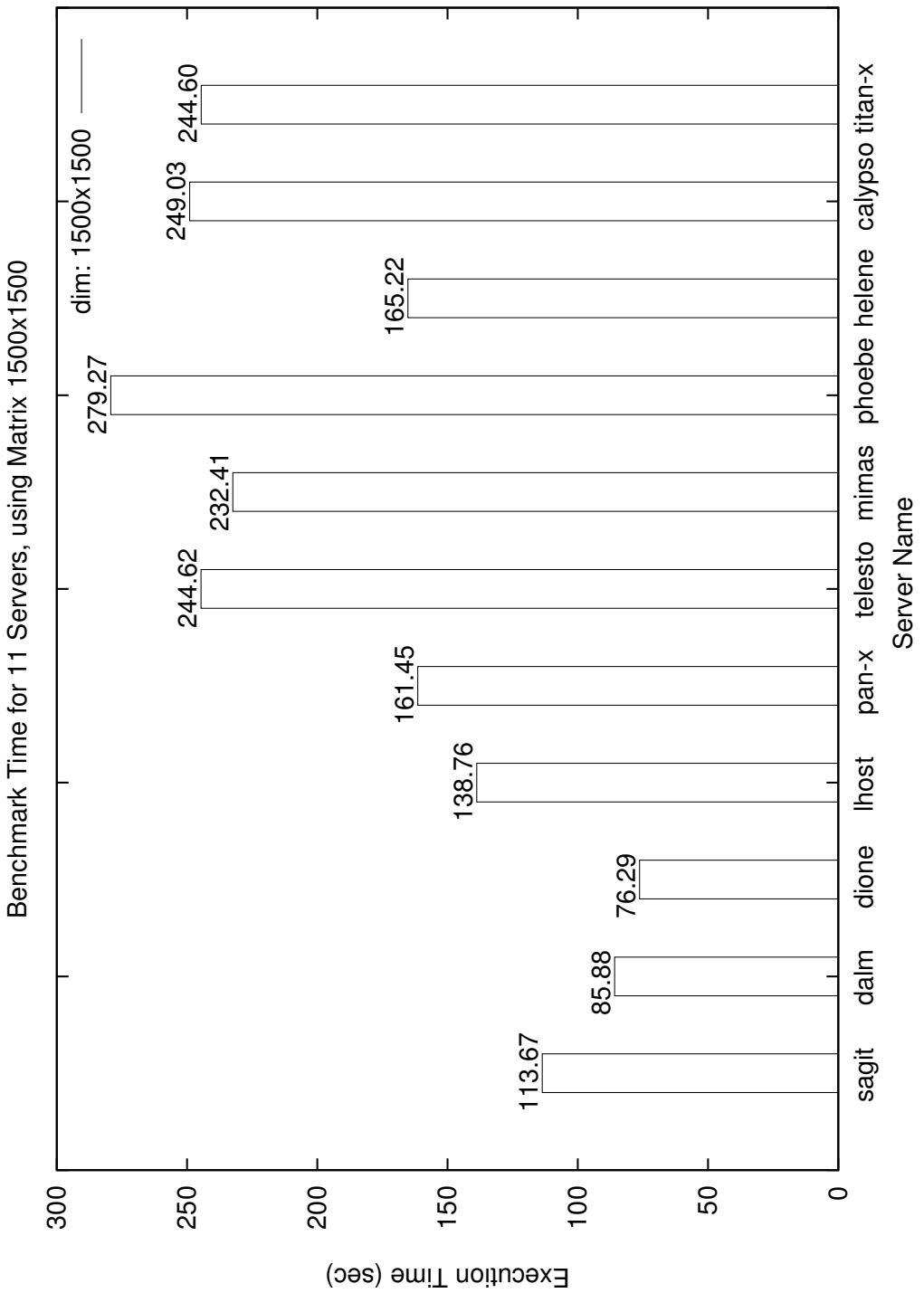


Figure 5.2: Matrix Benchmarking Results

`memory_free > 5`(MB). The details of the experiment is provided in Table. 5.3.

Item \ Library	Random	Smart Library
Matrix Size	1500×1500, blk=600	1500×1500. blk=600
No. of Servers	2	2
Requirement	*null*	$(host\_cpu\_bogomips > 4000)$ && $(host\_cpu\_free > 0.9)$ && $(host\_memory\_free > 5)$
Server List	lhost, phoebe	dalmatian, dione
Time used (sec)	100.16	63.00

Table 5.3: 2 vs 2 under zero Workload

The execution time for the two groups of servers were 100.16 seconds without using the Smart library and 63 seconds with the Smart library. The execution time was reduced by 37.1%.

2. **4 vs 4 without Workload**. Four servers were selected to compute the same 1500×1500 matrix with a block size of 200×200. From the benchmark step, users have the knowledge that P3 866MHz and P4 2.4GHz machines have better performance than P4 1.x GHz series. With this hint, experienced users may modify the server requirement to utilize P3 866MHz and P4 2.4GHz machines only. In Table. 5.4, the experiments details are given.

   By selecting the most suitable 4 servers out of the server pool, the execution time dropped from 62.61 seconds to 49.95 seconds, with an improvement of 20.2%.

3. **6 vs 6 without Workload**. In the third experiment, we made use of the *blacklist* option. The user specified the 5 servers which should

Item \ Library	Random	Smart Library
Matrix Size	1500×1500, blk=200	1500×1500. blk=200
No. of Servers	4	4
Requirement	*null*	$((host\_cpu\_bogomips > 4000)$ || $(host\_cpu\_bogomips < 2000))$ && $(host\_cpu\_free > 0.9)$ && $(host\_memory\_free > 5)$
Server List	phoebe, pandora-x, calypso, telesto	dalmatian, dione sagit, lhost
Time used (sec)	62.61	49.95

Table 5.4: 4 vs 4 under zero Workload

not be used during the computation. The 5 slowest servers from the benchmark list were eliminated as shown in the *Server List* entry in Table. 5.5.

Item \ Library	Random	Smart Library
Matrix Size	1500×1500, blk=200	1500×1500. blk=200
No. of Servers	6	6
Requirement	*null*	$(host\_cpu\_free > 0.9)$ && $(host\_memory\_free > 5)$ && $(user\_denied\_host1 = telesto)$ && $(user\_denied\_host2 = mimas)$ && $(user\_denied\_host3 = phoebe)$ && $(user\_denied\_host4 = calypso)$ && $(user\_denied\_host5 = titan\text{-}x)$
Server List	phoebe, pandora-x, calypso, telesto, helene, lhost	dalmatian, dione pandora-x, helene, lhost, sagit
Time used (sec)	46.90	43.02

Table 5.5: 6 vs 6 under zero Workload

As we can see, the matrix multiplication time was only reduced by 8.3%. The low improvement was caused by the increased communication overhead with 6 servers during computation and the large number of fast

servers selected in random set. Also, the same three hosts *pandora-x*, *helene*, and *lhost* were selected by both the random function and the Smart socket library, which further shortened the performance gap.

4. **4 vs 4 with Workload Enabled**. To measure the effects of the Smart socket library under non-zero workload, 7 servers with CPU P4 1.6GHz to 1.8 GHz were used to form the server pool. The program $Super\_PI$ was used to generate the workload. With given parameter 25, the $Super\_PI$ program will occupy 150 MBytes of memory and CPU usage will vary from 0% to 100%. The system load value will remain above 1. Out of the 7 servers, 3 of them were busy ones with $Super\_PI$ running, including *helene*, *telesto* and *mimas*. The experiment comparison is given in Table. 5.6.

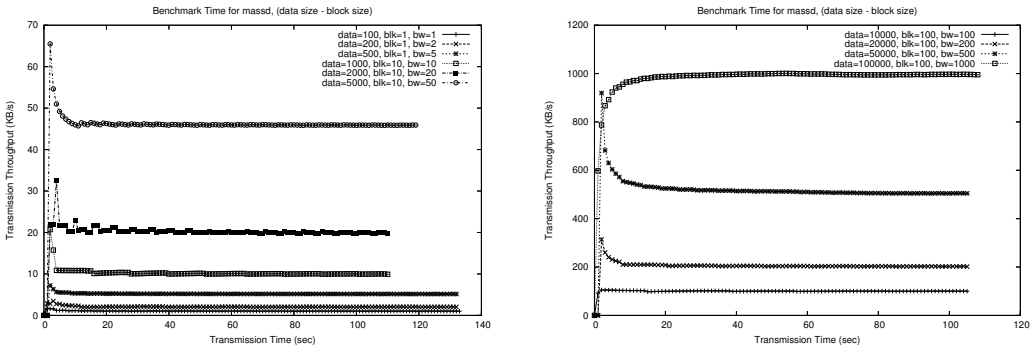
Item \ Library	Random	Smart Library
Matrix Size	1500×1500, blk=200	1500×1500. blk=200
No. of Servers	4	4
Requirement	*null*	$(host\_cpu\_free > 0.9)$ && $(host\_memory\_free > 5)$ && $(host\_system\_load1 < 0.5)$
Server List	mimas, helene, calypso, telesto	calypso, phoebe, titan-x, pandora-x
Time used (sec)	90.93	66.72

Table 5.6: 4 vs 4 with Workload

By avoiding the 3 busy servers, the matrix computation was completed in 66.72 seconds, compared with 90.93 seconds of the 4 randomly selected servers. The improvement was 26.6%.

5.3.2 Massive Download

We also developed a massive download program *massd* by using the same algorithm as the matrix multiplication program. The *massd* program can download data from multiple servers simultaneously. The average throughput of the massive download program was measured as the performance indicator. *Rshaper*[rshaper01] was used to set the link bandwidth to a random value, simulating the conditions on a real network. To verify the smooth cooperation between *massd* and *rshaper*, we ran 10 sample tests. The result in Fig. 5.3, shows that the maximum throughput that can be achieved by *massd* can be precisely controlled by *rshaper*.



(a) (b)

Figure 5.3: Benchmark for *rshaper* and *massd*

Each transmission was done by using the parameter `(data, blk, blk)`. Parameter `data` refers to the total amount of data to transmit and `blk` is the size of basic block allocated to each server; the unit is KBytes. The third parameter `bw` is the bandwidth value set by *rshaper* and was set to 1% of `data`; the unit is KBytes/sec. From Fig. 5.3, we can see the bandwidth

values set by *rshaper* were very close to the actual throughput we can get from *massd* program. This means the overhead of the programs used in the experiments has negligible side effects.

In the formal experiments, we selected 6 machines as the file servers: *mimas*, *telesto*, *lhost* in group-1 and *dione*, *titan-x*, *pandora-x* in group-2. All machines in the same group were assigned the same network bandwidth by *rshaper* in the range from 0 Mbps to 10 Mbps. The bandwidth value was generated randomly, so that the two groups of servers had different bandwidth values. The group with the higher bandwidth is called the fast server group and the other group is the slow server group. The transmission throughput from the fast server group should be higher than the slow server group. In the conventional socket library, users have to randomly select servers, without the help from third-party utilities. By using the Smart socket library, users can pick up the fast servers for data transmission with high throughput by providing the proper requirement specification.

For comparison, 3 sets of experiments were done with 1, 2 and 3 file servers used in each. The data size is 50000 KBytes and the block size is 100 KBytes.

1. **1 Server for massd**. The experiment information is given in Table. 5.7. The bandwidth assigned to group-1 servers is 6.72 Mbps and group-2 servers have bandwidth of 1.33 Mbps. The amount of data to transmit is 50000 KBytes. The randomly selected server is *pandora-x* from group-1. With server requirement `monitor_network_bw > 6`, only server with bandwidth larger than 6 Mbps will be selected by the

Smart socket library. The machine *lhost* was selected.

Item	Value
Group-1 bandwidth	6.72 Mbps
Group-2 bandwidth	1.33 Mbps
Random Servers	*pandora-x*
Smart Servers	*lhost*
Server Req	$monitor\_network\_bw > 6$
Transmission Data	50000 KB by 100 KB

Table 5.7: Experiment for 1vs1 massd

The throughput comparison is given in Fig. 5.4. We can see that with the help of the Smart socket library function, the optimal server was used. The throughput was increased from 170 KB/s to 860 KB/s.

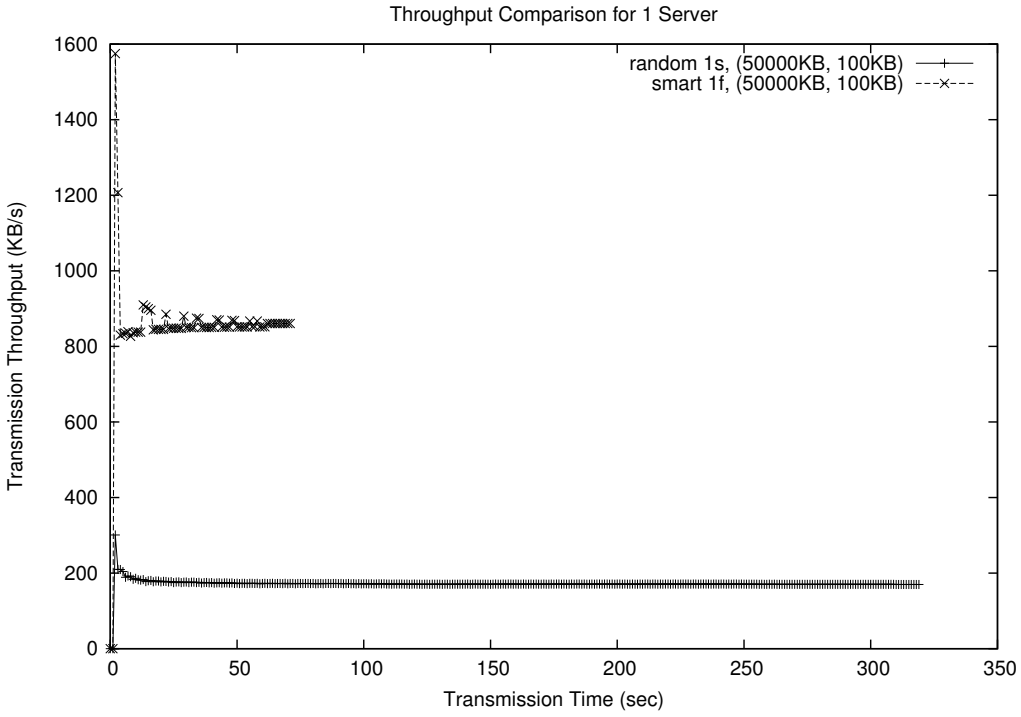


Figure 5.4: Experiments for massd: 1 vs 1

2. **2 Servers for massd**. When two servers are required, 3 types of

choices are possible: two slow servers, 1 slow server plus 1 fast server and two fast servers. When the random function chooses two optimal servers, the performance will be similar for both random function and the Smart library. In other cases, the Smart library will provide better results. The experiment details are listed in Table. 5.8.

Item	Value
Group-1 bandwidth	5.01 Mbps
Group-2 bandwidth	7.67 Mbps
Random1 Servers	*mimas, telesto*
Random2 Servers	*telesto, titan-x*
Smart Servers	*titan-x, pandora-x*
Server Req	$monitor\_network\_bw > 7$
Transmission Data	50000 KB by 100 KB

Table 5.8: Experiment for 2vs2 massd

In this experiment, group-2 had a higher bandwidth of 7.67 Mbps. With the Smart library, 2 servers were picked up from this group, *titan-x* and *pandora-x*. The two random server sets contain zero fast server and one fast server each. The performance chart is shown in Fig. 5.5.

The first random set with no fast servers achieved an average throughput of 660 KB/s and the second random set with one fast server had a throughput of 795 KB/s. Both values are lower than the what we got by using the Smart library, which is 994 KB/s.

3. **3 Servers for massd**. In the last set of experiments, 3 data servers were used. There are 4 possible combinations: 4 groups of servers with 0, 1, 2 and 3 fast servers each. We call the 4 combinations: random set-1, random set-2, random set-3 and the smart set. The servers used

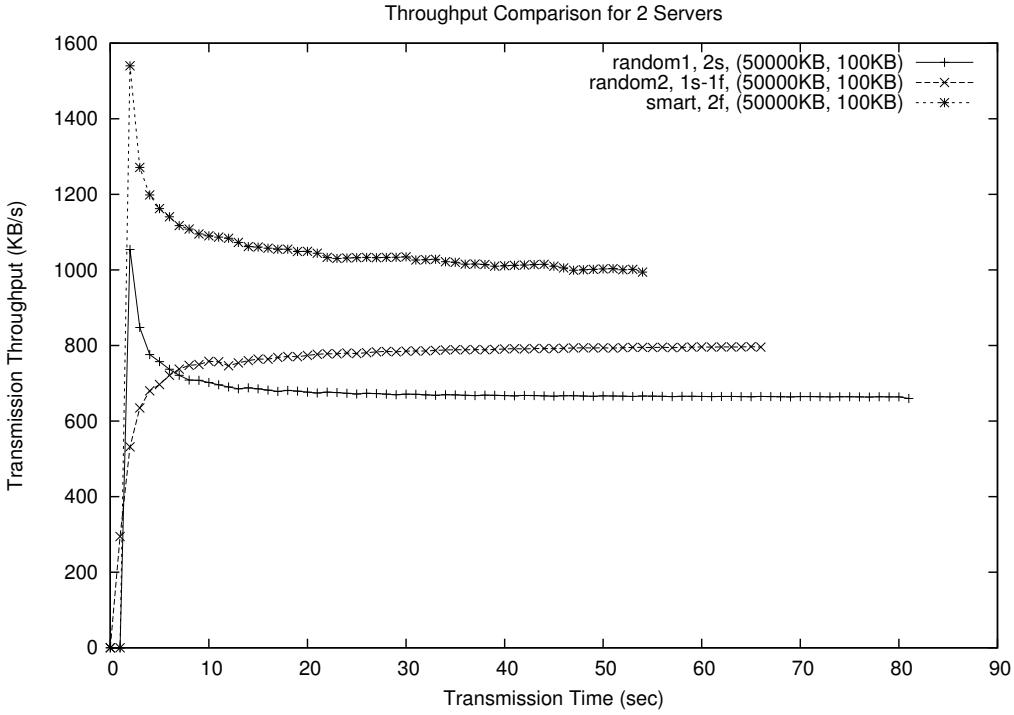


Figure 5.5: Experiments for massd: 2 vs 2

for each one of them are listed in Table. 5.9.

Fig. 5.6 illustrates the performance results for the 4 transmissions. The throughput values are 387 KB/s, 520 KB/s, 634 KB/s and 796 KB/s for random set-1, random set-2, random set-3 and Smart set. The transmission with the assistance of the Smart socket library experienced the highest throughput as expected.

In this chapter, we presented experiment procedures and performance analysis of the new socket library. In the next chapter, we will discuss some possible improvements that can be done in the future work.

Item	Value
Group-1 bandwidth	5.99 Mbps
Group-2 bandwidth	2.92 Mbps
Random1 Servers	*dione, titan-x, pandora-x*
Random2 Servers	*mimas, titan-x, dione*
Random3 Servers	*telesto, mimas, dione*
Smart Servers	*lhost, telesto, mimas*
Server Req	$monitor\_network\_bw > 5$
Transmission Data	50000 KB by 100 KB

Table 5.9: Experiment for 3vs3 massd

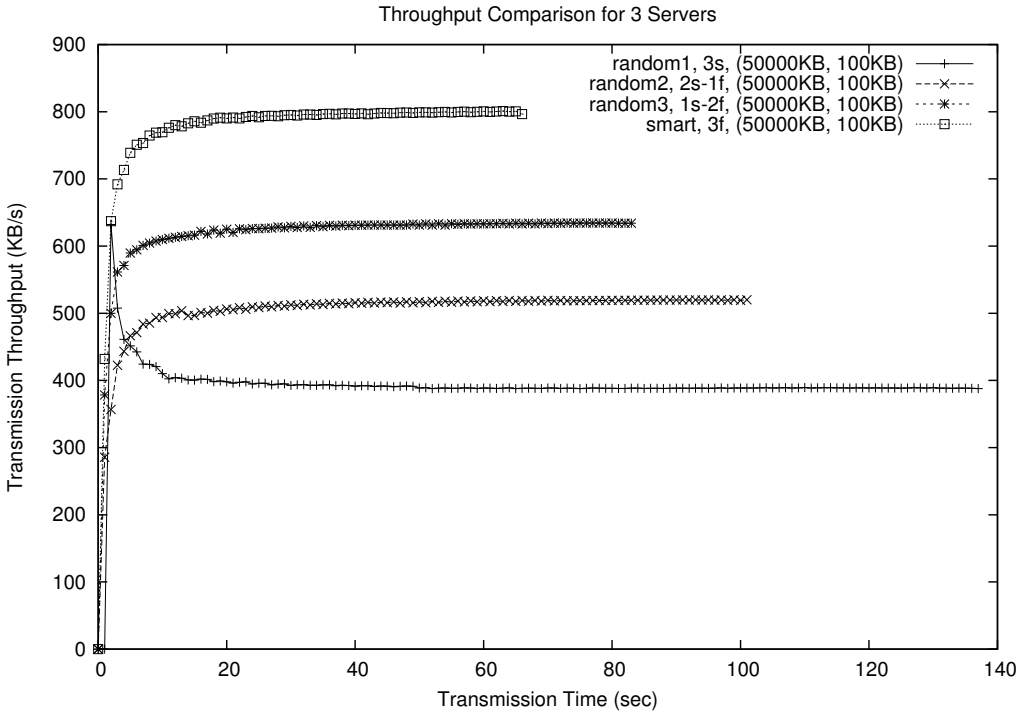


Figure 5.6: Experiments for massd: 3 vs 3

Chapter 6

Future Work

Despite of the various features, the current implementation of the Smart TCP socket still consists of a few limitations. In this chapter, we suggest below some possible improvements and extensions, in order to make this library more applicable.

- Fault-tolerance. Error recovery should be handled at the application level, as it involves process check-pointing and resumption. For the network layer, we will look at the complications during the data transmission. A new set of socket functions will be added to suspend and resume the sockets, such that the program recovery and process migration steps can be done more smoothly. The reliable socket library *rsocks*[rsocks01] is working at this area.

- Task division module. The Smart TCP socket library can perform better if the task division module is intelligent enough to automatically decide the proper server requirement for each task and requests for

sockets accordingly. This requires the task division module to examine the program code and arrange the optimal resource usage. Finding schemes on task division are still under active research and a lot of work can be done towards this direction.

- Integration into the kernel. Currently the Smart TCP socket library is developed and experimented at the application level. In application level it is easier to make changes and debug, yet the overhead of these library calls could be high. After we stabilize the code of the Smart TCP socket, we can make an attempt to integrate it into the kernel level. As the outcome, applications would experience less overhead and fast response.

- Selected parameters. By default, the server probes measure all defined parameters and report them back to the system monitor. If the number of parameters grows high, it will lead to measurement time, higher system workload and higher network bandwidth usage. For a normal application, it is unlikely that all system resources will be required at the same priority level. Generally, only a small subset of the parameters are of concern for a particular application. The wizard and the server monitor can be modified to summarize the most popular system parameters and inform the probes to report only those parameters that are mostly of concern. Currently, all the server attributes have numerical values. In later development, we may need to add in attributes with string values in order to parse statements like `"machine_type=i386"`.

- UDP vs TCP. Because of the low overhead, the UDP protocol is used to

transmit the server status report. For short server report and normal network connections, it will be sufficient. For long server reports under congested networks, some UDP packets may get lost, which makes the server status unusable. In that situation, the probes should be instructed to use TCP for reporting.

- Server report issues. In this thesis, we assume that all servers provide identical services in a controlled environment. However, in an actual distributed computing environment, different servers may offer distinct services. We can extend the function of the server probe and allow it to report the types of services available on every server. Also The *wizard* program examines the server reports one by one, which makes it very difficult for users to write a requirement like `"3 servers with largest memory"`. The wizard needs to be modified to check multiple server reports for one requirement at a time instead of sequential scanning.

Chapter 7

Conclusion

The middle-ware is becoming more complicated, particularly in the context of GRID. One of the causes for this complexity is the server selection part. This project tackles the problem at the network layer. We have presented a Smart TCP socket library for the distributed computing environment, allowing users to specify what is the best for the application and select the best servers according to user requirement. With the help of our Smart TCP socket library, users can focus on task division process and use the set of sockets returned to handle each segment of the task. The new library provides the following features:

High-level programming interface It provides an easy programming interface for users to write network applications in a server-controlled environment. The application is not compelled to be aware of the domain names or IP addresses of servers. All a user needs to tell is the service type for the application and the server resource requirement. The server resource requirement can be specified by using a meta lan-

guage defined for sophisticated mathematical expressions.

Convenience for server selection algorithms The server probe measures a full range of system status parameters, from CPU usage rate, memory space, hard disk IO to network bandwidth and send back the status report to the server monitor. The abundant parameters being probed can help users to develop new server selection schemes based on resource monitoring.

This mechanism separates the server selection module out of the middleware and integrate it into the socket level. That will make the new middle-ware less complicated and greatly reduce the servers' workload, when multiple distributed applications using different probing-based middle-wares are required on the same machine. The same set of server probes, monitors and wizard can be used smoothly, as long as these middle-wares use the same interfaces to communicate with the Smart socket layer. The same copy of server reports could be used by different middle-ware decision modules and different algorithms can be applied.

Real time report from servers The available servers periodically send reports back to the monitor. The dynamically generated reports can help middle-wares to make good decisions about which servers to use. Since it reflects the actual server workload at real time, the selected servers should generate much better performance than those selected based on fixed server configuration files, especially under heavy load.

In case of a server failure, the monitor can easily detect it, remove the failed node from the server pool and prevents subsequent tasks

from being assigned to the failed server. This is also the first step for fault-recovery implementation, that may redirect the failed connection to other running servers to resume the task. However, the checkpoint function, and the recovery procedure should be accomplished in the upper level.

Expandable framework A standard procedure for adding the host side and user side parameters has been established. New parameters can be added in the same way and new decision making algorithms can use those new parameters immediately, according to users' decision.

The Smart socket library is built upon the standard BSD socket library and the inter-process communication part follows the classic System-V standard. Both of these two system libraries are supported in most of today's popular UNIX systems. Also as the whole package is developed in the user space, the Smart TCP socket library can be used in most UNIX or UNIX-derived systems without any modification.

In conclusion, the Smart TCP socket layer is an attempt to influence new changes in the GRID middle-ware design. If we can standardize the format of the server status reports and the library interfaces, we can integrate the system resource monitoring function into the network layer. This will allow multiple middle-ware implementations to co-exist without introducing extra server load and network traffic. The new socket interface enables the middle-ware designers to focus on improving the task scheduler function and thus encourages the popularity of GRID computing facilities.

Bibliography

[alkindi00] "Run-time Optimisation Using Dynamic Performance Prediction", A. M. Alkindi, D. J. Kerbyson, E. Papaefstathiou, G. R. Nudd, High Performance Computing and Networking, LNCS, Vol. 1823, Springer-Verlag, May 2000, pp. 280-289.

[ants04] "The ANTS Load Balancing System", Jakob Østergaard, *http://unthought.net/antsd/info.html*, 2004.

[brian84] "The UNIX Programming Environment", Brian W. Kernighan and Rob Pike, Prentice Hall, 1984.

[carter96] Robert L. Carter, Mark E. Crovella, *Measuring Bottleneck Link Speed in Packet-Switched Networks*, *Performance Evaluation* Vol. 27&28, 1996.

[cisco04] "Cisco NAC: The Development of the Self-Defending Network", *http://www.cisco.com/warp/public/cc/so/neso/sqso/csdni_wp.htm*, Cisco Systems, Inc. 2004.

[condor04] "Condor Project", CS Department, UW-Madison, *http://www.cs.wisc.edu/condor/*.

[constantinos01] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore *What do packet dispersion techniques measure?*, Inforcom 2001, Anchorage Alaska USA, 2001

[erik01] "Linux Kernel Procfs Guide", Erik(J. A. K) Mouw, *http://www.kernelnewbies.org/documents/kdoc/procfs-guide/lkprocfsguide.html*, 2001.

[gfi04] "GFI LANguard Network Security Scanner 5 Manual", *http://www.gfi.com/lannetscan*, GFI Software Ltd., 2004.

[geist96] "PVM and MPI: a Comparison of Features", G. A. Geist, J. A. Kohl, P. M. Papadopoulos, May 30, 1996, *http://www.csm.ornl.gov/pvm/PVMvsMPI.ps.*

[fyodor98] "Remote OS detection via TCP/IP Stack FingerPrinting", *http://www.insecure.org/nmap/nmap-fingerprinting-article.html*, Fyodor, 1998.

[globus04] "Globus Alliance", *http://www.globus.org/.*

[gnuflex00] "Flex: A fast lexical analyser generator", *http://www.gnu.org/software/flex/*, 2000.

[gnubison03] "Bison: A general-purpose parser generator", *http://www.gnu.org/software/bison/*, 2003.

[gnuproject04] "GNU Operating System - Free Software Foundation", *http://www.gnu.org/*, 2004.

[kurose03] James F. Kurose, Keith W. Ross, *Computing Networking: A Top-Down Approach Featuring the Internet*, Addison Wesley 2003.

[lexyacc92] "Lex & Yacc" 2_{nd} edition, John R. Levine, Tony Mason, Doug Brown, O'Reilly & Associates, 1992.

[libpcap04] "libpcap", Lawrence Berkeley National Laboratory, Network Research Group, emphhttp://www-nrg.ee.lbl.gov/

[lvserver04] "Linux Virtual Server Project", *http://www.linuxvirtualserver.org/.*

[manish02] Manish Jain, Constantinos Dovrolis *"End-to-End Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput"*, ACM SIGCOMM 2002, Pittsburgh PA USA, 2002.

[manish02pl] "Pathload: a measurement tool for end-to-end available bandwidth", Manish Jain, Constantinos Dovrolis, PAM 2002.

[mpi04] "The Message Passing Interface (MPI) standard", MCS Division, Argonne National Laboratory, *http://www-unix.mcs.anl.gov/mpi/.*

[ncs03] "Network Characterization Service (NCS)", Computational Research Division, Lawrence Berkeley National Laboratory, *http://www-didc.lbl.gov/NCS/*, 2003.

[openmosix04] "OpenMosix", *http://openmosix.sourceforge.net/.*

[ogsa04] "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", Ian Fos-

ter, Carl Kesselman, Jeffrey M. Nick, Steven Tuecke, *http://www.globus.org/research/papers/ogsa.pdf.*

[pvm04] "Parallel Virtual Machine", Computer Science and Mathematics Division, Oak Ridge National Laboratory, *http://www.csm.ornl.gov/pvm/pvm_home.html*, 2004.

[p4system93] "Monitors, messages, and clusters: The p4 parallel programming system", R. Butler and E. Lusk, Technical Report Preprint MCS-P362-0493, Argonne National Laboratory, 1993.

[rajesh98] "Matchmaking: Distributed Resource Management for High Throughput Computing", Rajesh Raman, Miron Livny, and Marvin Solomon, HPDC-98, 1998.

[rshaper01] "rshaper", Allessandro Rubini, *http://ar.linux.it/software/*, Nov 2001.

[rsocks01] "Reliable Sockets", Victor C. Zandy, Barton P. Miller, *http://www.cs.wisc.edu/ zandy/rocks/*, 2001.

[shaotao03] Shao Tao, L. Jacob, A. L. Ananda *"A TCP Socket Buffer Auto-tuning Daemon"*, ICCCN 2003, Dallas TX USA, 2003.

[steve01] Steve Steinke, *Network Delay and Signal Propogation*, *http://www.networkmagazine.com/article/NMG20010416S0006.*

[superpi04] "Super PI", Kanada Laboratory, *http://pi2.cc.u-tokyo.ac.jp/.*

Appendix A

Pipechar results

A.1 from *sagit* to *cmui*

```
sagit:/home/shaotao/master/ver_2/test/raw_socket# ./pipechar cmui
0: localhost [23 hops] ()                    min forward time, min RTT, avg RTT
 1: gw-a-15-810.comp.nus.edu.sg         (137.132.81.6)    0.75   0.20   2.39ms
 2: NoNameNode                          (192.168.15.6)    0.74   0.40   2.36ms
 3: 115-18.priv.nus.edu.sg              (172.18.115.18)   0.74   0.50   2.26ms
 4: core-s15-vlan142.priv.nus.edu.sg    (172.18.20.125)   0.73   0.60   2.47ms
 5: core-au-vlan51.priv.nus.edu.sg      (172.18.20.13)    0.75   0.60   2.23ms
 6: svrfrm1-cc-vlan167.priv.nus.edu.sg  (172.18.20.98)    0.73   0.60   2.41ms
 7: border-pgp-m1.nus.edu.sg            (137.132.3.131)   1.33 374.20 363.58ms
 8: ge3-12.pgp-dr1.singaren.net.sg      (202.3.135.129)  26.57 362.30 322.72ms
32 bad fluctuation
 9: ge3-0-2.pgp-cr1.singaren.net.sg     (202.3.135.17)   -1.62 378.50 385.91ms
10: pos1-0.seattle-cr1.singaren.net.sg  (202.3.135.5)    36.12 539.40 478.77ms
32 bad fluctuation
11: Abilene-PWAVE-1.peer.pnw-gigapop.net(198.32.170.43) -21.94 524.20 509.76ms
12: dnvrng-sttlng.abilene.ucaid.edu     (198.32.8.50)     7.07 547.30 524.40ms
13: kscyng-dnvrng.abilene.ucaid.edu     (198.32.8.14)     7.26 552.30 504.06ms
14: iplsng-kscyng.abilene.ucaid.edu     (198.32.8.80)    15.84 562.00 523.04ms
15: chinng-iplsng.abilene.ucaid.edu     (198.32.8.76)    16.59 547.60 507.34ms
32 bad fluctuation
16: nycmng-chinng.abilene.ucaid.edu     (198.32.8.83)    -4.51 597.70 543.96ms
17: washng-nycmng.abilene.ucaid.edu     (198.32.8.85)    13.15 566.70 567.81ms
18: beast-abilene-p3-0.psc.net          (192.88.115.125)          0.04 618.70    nanms
19: bar-beast-ge-0-1-0-1.psc.net        (192.88.115.17)   0.33 554.40 522.08ms
20: cmu-i2.psc.net                      (192.88.115.186)          9.99 585.60 501.58ms
32 bad fluctuation
21: CORE0-VL501.GW.CMU.NET              (128.2.33.226)   -5.86 618.80 468.77ms
22: CS-VL1000.GW.CMU.NET                (128.2.0.8)      36.98 640.40 464.34ms
32 bad fluctuation
23: cmui                                (128.2.220.137) -343.88 591.50 454.09ms

PipeCharacter statistics: 66.17% reliable
From localhost:
|       96.644 Mbps  100BT (102.9328 Mbps)

1: gw-a-15-810.comp.nus.edu.sg     (137.132.81.6)
|
|       158.757 Mbps             <1.2081% BW used>
2: NoNameNode                       (192.168.15.6)
```

```
|
|       100.730 Mbps             <0.0000% BW used>
3: 115-18.priv.nus.edu.sg          (172.18.115.18)
|
|       159.374 Mbps             <0.9511% BW used>
4: core-s15-vlan142.priv.nus.edu.sg(172.18.20.125)
|
|       162.706 Mbps             <3.0585% BW used>
5: core-au-vlan51.priv.nus.edu.sg  (172.18.20.13)
|
|       156.608 Mbps             <2.3936% BW used>
6: svrfrm1-cc-vlan167.priv.nus.edu.sg(172.18.20.98)
|
|       151.314 Mbps    !!!      <94.9947% BW used>
7: border-pgp-m1.nus.edu.sg        (137.132.3.131)
|
|       9.687 Mbps    !!!        <72.9038% BW used> May get 94.99% congested
8: ge3-12.pgp-dr1.singaren.net.sg  (202.3.135.129)
| hop analyzed: 0.77 : 0.00
|
|       0.755 Mbps    !!! ??? congested bottleneck <46.8769% BW used>
9: ge3-0-2.pgp-cr1.singaren.net.sg (202.3.135.17)
| hop analyzed: 0.51 : 8.39
|
|       9.934 Mbps    !!!        <90.8149% BW used>
10: pos1-0.seattle-cr1.singaren.net.sg(202.3.135.5 )
| hop analyzed: 0.96 : 0.00
|
|       0.948 Mbps    !!! ??? congested bottleneck <90.3784% BW used>
11: Abilene-PWAVE-1.peer.pnw-gigapop.net(198.32.170.43)
|
|       9.590 Mbps    !!!        <90.5481% BW used>
12: dnvrng-sttlng.abilene.ucaid.edu (198.32.8.50 )
|
|       10.071 Mbps              <2.5222% BW used>
13: kscyng-dnvrng.abilene.ucaid.edu (198.32.8.14 )
|
|       10.132 Mbps              <4.5150% BW used>
14: iplsng-kscyng.abilene.ucaid.edu (198.32.8.80 )
| hop analyzed: 0.86 : 21.25
|
|       43.138 Mbps    !!!       <78.6142% BW used>
15: chinng-iplsng.abilene.ucaid.edu (198.32.8.76 )
| hop analyzed: 1.05 : 1.36
|
|       1.350 Mbps    !!! ??? congested bottleneck <86.3983% BW used>
16: nycmng-chinng.abilene.ucaid.edu (198.32.8.83 )
|
|       5.292 Mbps    !!! ??? congested bottleneck <99.7811% BW used>
17: washng-nycmng.abilene.ucaid.edu (198.32.8.85 )
| hop analyzed: 0.00 : 2250.00
|
|       2477.365 Mbps    !!!     <99.7567% BW used>
18: beast-abilene-p3-0.psc.net      (192.88.115.125)
|
|       970.166 Mbps    !!!      <78.2493% BW used> May get 90.33% congested
19: bar-beast-ge-0-1-0-1.psc.net    (192.88.115.17)
|
|       9.851 Mbps    !!!        <27.9351% BW used> May get 96.69% congested
20: cmu-i2.psc.net                  (192.88.115.186)
| hop analyzed: 1.30 : 0.00
|
```

```
|       1.479 Mbps               <10.3610% BW used> May get 81.96% congested
21: CORE0-VL501.GW.CMU.NET          (128.2.33.226)
| hop analyzed: 1.01 : 0.00
|
|       1.434 Mbps    !!!        <30.1162% BW used> May get 22.04% congested
22: CS-VL1000.GW.CMU.NET            (128.2.0.8   )
|       -0.209 Mbps *** static bottle-neck possible modern (0.5637 Mbps)

23: cmui                            (128.2.220.137)
```

A.2 from *sagit* to *tokxp*

```
sagit:/home/shaotao/master/ver_2/test/raw_socket# ./pipechar tokxp
0: localhost [15 hops] ()                    min forward time, min RTT, avg RTT
 1: gw-a-15-810.comp.nus.edu.sg         (137.132.81.6)    0.74   0.20   2.12ms
 2: NoNameNode                          (192.168.15.6)    0.74   0.40   2.13ms
 3: 115-18.priv.nus.edu.sg              (172.18.115.18)   0.74   0.50   2.71ms
 4: core-s15-vlan142.priv.nus.edu.sg    (172.18.20.125)   0.74   0.60   2.47ms
 5: core-au-vlan51.priv.nus.edu.sg      (172.18.20.13)    0.74   0.60   2.56ms
 6: svrfrm1-cc-vlan167.priv.nus.edu.sg  (172.18.20.98)    0.74   0.60   2.59ms
 7: border-pgp-m1.nus.edu.sg            (137.132.3.131)   0.72   1.10   2.93ms
 8: ge3-12.pgp-dr1.singaren.net.sg      (202.3.135.129)   0.72   1.20   2.78ms
 9: fe4-1-0101.pgp-ihl1.singaren.net.sg (202.3.135.34)    0.79   1.40   3.11ms
10: atm3-040.pgp-sox.singaren.net.sg    (202.3.135.66)    0.72   1.80   3.62ms
11: ascc-gw.sox.net.sg                  (198.32.141.28)   0.82   1.90   3.46ms
12: s1-1-0-0.br0.tpe.tw.rt.ascc.net     (140.109.251.74)          0.70  48.80  52.06ms
13: s4-0-0-0.br0.tyo.jp.rt.ascc.net     (140.109.251.41)          0.76  78.60  87.67ms
14: tpr2-ae0-10.jp.apan.net             (203.181.248.154)         0.68 126.30 139.56ms
15: tokxp                               (203.181.248.24)          0.04 126.70 128.28ms

PipeCharacter statistics: 97.70% reliable
From localhost:
|       97.561 Mbps  100BT (97.0672 Mbps)

1: gw-a-15-810.comp.nus.edu.sg     (137.132.81.6)
|
|       151.243 Mbps             <0.8064% BW used>
2: NoNameNode                       (192.168.15.6)
|
|       99.270 Mbps              <0.1344% BW used>
3: 115-18.priv.nus.edu.sg           (172.18.115.18)
|
|       150.626 Mbps             <0.0000% BW used>
4: core-s15-vlan142.priv.nus.edu.sg(172.18.20.125)
|
|       147.294 Mbps             <0.2692% BW used>
5: core-au-vlan51.priv.nus.edu.sg  (172.18.20.13)
|
|       153.392 Mbps             <0.8097% BW used>
6: svrfrm1-cc-vlan167.priv.nus.edu.sg(172.18.20.98)
|
|       151.314 Mbps             <2.7211% BW used>
7: border-pgp-m1.nus.edu.sg         (137.132.3.131)
|
|       153.667 Mbps             <0.9695% BW used>
8: ge3-12.pgp-dr1.singaren.net.sg  (202.3.135.129)
|
|       152.342 Mbps             <9.0680% BW used>
9: fe4-1-0101.pgp-ihl1.singaren.net.sg(202.3.135.34)
```

```
|
|       152.710 Mbps             <9.0680% BW used>
10: atm3-040.pgp-sox.singaren.net.sg(202.3.135.66)
|
|       151.983 Mbps             <12.1655% BW used>
11: ascc-gw.sox.net.sg              (198.32.141.28)
|
|       153.250 Mbps             <14.3550% BW used>
12: s1-1-0-0.br0.tpe.tw.rt.ascc.net (140.109.251.74)
|
|       152.537 Mbps             <7.4891% BW used>
13: s4-0-0-0.br0.tyo.jp.rt.ascc.net (140.109.251.41)
|
|       104.591 Mbps    !!! ??? congested bottleneck <95.7833% BW used>
14: tpr2-ae0-10.jp.apan.net         (203.181.248.154)
|       1800.000 Mbps  OC48 (2481.9865 Mbps)

15: tokxp                           (203.181.248.24)
```

A.3 from *sagit* to *suna*

```
sagit:/home/shaotao/master/ver_2/test/raw_socket# ./pipechar suna
0: localhost [3 hops] ()                    min forward time, min RTT, avg RTT
 1:  1.9s gw-a-15-810.comp.nus.edu.sg(137.132.81.6)       0.76   0.20   2.16ms
 2:  1.4s sf0.comp.nus.edu.sg (137.132.90.52)     0.76   0.50   2.82ms
 3:  2.9s sf0.comp.nus.edu.sg (137.132.90.52)     0.74   0.50   2.31ms

PipeCharacter statistics: 95.05% reliable
From localhost:
|       95.238 Mbps  100BT (102.9328 Mbps)

1: gw-a-15-810.comp.nus.edu.sg     (137.132.81.6)
|
|       100.716 Mbps             <0.1321% BW used>
2: sf0.comp.nus.edu.sg             (137.132.90.52)
|       96.774 Mbps  100BT (100.7303 Mbps)

3: sf0.comp.nus.edu.sg             (137.132.90.52)
```

Appendix B

Keywords and Functions

B.1 Server-side Variables

```
‘‘host_system_load1’’, 0,
‘‘host_system_load5’’, 0,
‘‘host_system_load15’’, 0,

‘‘host_cpu_bogomips’’, 0,

‘‘host_cpu_user’’, 0,
‘‘host_cpu_nice’’, 0,
‘‘host_cpu_system’’, 0,
‘‘host_cpu_free’’, 0,

‘‘host_memory_total’’, 0,
‘‘host_memory_used’’, 0,
‘‘host_memory_free’’, 0,

‘‘host_disk_allreqps’’, 0,
‘‘host_disk_rreqps’’, 0,
‘‘host_disk_rblocksps’’, 0,
‘‘host_disk_wreqps’’, 0,
‘‘host_disk_wblocksps’’, 0,

‘‘host_network_rbytesps’’, 0,
‘‘host_network_rpacketsps’’, 0,
‘‘host_network_tbytesps’’, 0,
‘‘host_network_tpacketsps’’, 0,
```

```
‘‘monitor_network_bw", -1
‘‘monitor_network_rtt", -1
```

B.2 User-side Variables

```
‘‘user_picked_host1’’, -1,
‘‘user_picked_host2’’, -1,
‘‘user_picked_host3’’, -1,
‘‘user_picked_host4’’, -1,
‘‘user_picked_host5’’, -1,

‘‘user_denied_host1’’, -1,
‘‘user_denied_host2’’, -1,
‘‘user_denied_host3’’, -1,
‘‘user_denied_host4’’, -1,
‘‘user_denied_host5’’, -1,
```

B.3 Constants

```
‘‘PI’’,       3.1415926,
‘‘E’’,        2.7182818,
‘‘GAMMA’’,    0.5772156,
‘‘DEG’’,     57.2957795,
‘‘PHI’’,      1.6180339 ,
```

B.4 Math Functions

```
‘‘sin’’,  sin,
‘‘cos’’,  cos,
‘‘atan’’, atan,
‘‘log’’,  Log,
‘‘log10’’,    Log10,
‘‘exp’’,  Exp,
‘‘sqrt’’, Sqrt,
‘‘int’’,  integer,
‘‘abs’’,  fabs,
```

Appendix C

Experiment Programs

C.1 Distributed Matrix Multiplication

The matrix multiplication program contains both a local computation mode and a distributed computation mode. In the local mode, the two input matrices are multiplied in the vector multiplication style and the result entries are recorded into the output matrix. The local mode multiplication algorithm is given in Algorithm. 1.

Algorithm 1 Matrix Multiplication in Local mode

1: dim - square matrix dimension size
2: $matrix_A$ - first input matrix
3: $matrix_B$ - second input matrix
4: $matrix_C$ - output matrix
5: **for** $i = row_0$ to row_{dim-1} **do**
6: **for** $j = col_0$ to col_{dim-1} **do**
7: $matrix_C[i][j] = \sum_{t=0}^{dim-1} matrix_A[i][t] \times matrix_B[t][j]$
8: **end for**
9: **end for**

The diagram for showing how the computation is in Fig. C.1.

For every entry in the result matrix $Matrix_C$, we need one *slice* from input matrix $Matrix_A$ and another *slice* from the second input $Matrix_B$. We call the two slices $Slice_A$ and $Slice_B$. For the *local* mode, *width* of the slices will be 1. For the *Distributed* mode, the dimension of the slices will be based on the parameters given in the scenario including: the matrix dimension *dim*, the block dimension *blkdim*. The parameter *blkdim* is the size of a unit block in $Matrix_C$. Thus, assuming the difference between row_1 and row_2 or col_1 and col_2 is *delta*, the value of *delta* would be:

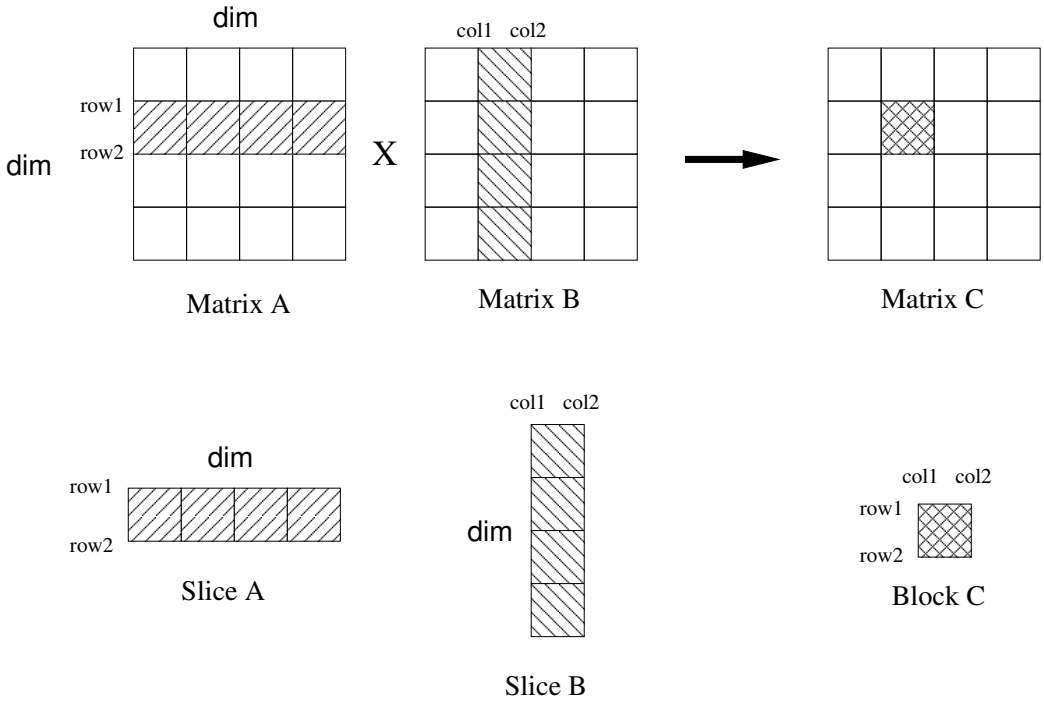


Figure C.1: Matrix Multiplication

$$delta = \begin{cases} dim & \text{if } blkdim \geq dim \\ blkdim & \text{if } blkdim < dim \text{ and not last row/column} \\ blkdim & \text{if } blkdim < dim, \text{ last row/column, dim\%blkdim} = 0 \\ dim\%blkdim & \text{if } blkdim < dim, \text{ last row/column, } dim\%blkdim \neq 0 \end{cases}$$

The total number of blocks created for $Matrix_C$ in the *Distributed* mode is $nblock = (\lceil \frac{dim}{blkdim} \rceil)^2$. Each block will be identified by a vector structure ($blkid$, row_1, row_2, col_1, col_2). The $blkid$ is the sequence number for a block ranging from 0 to $nblock - 1$. (row_1, row_2) is the identification of $Slice_A$ and (col_1, col_2) is the identification of $Slice_B$. With such a *block* structure, the matrix multiplication can be considered as computing a group of small matrix blocks, each one independent from another.

The distributed computation is accomplished by the *master* program and *worker* programs. A *master* program running on client machine assigns the block tasks to *workers* and collects the returned results. The *worker* programs running on the server machines will receive the slices and block structure, compute $Block_C$ and send back the result.

The matrix blocks ($Slice_A$, $Slice_B$, $Block_C$) will be sent to the available servers sequentially. Depending on the configuration of the servers and the

block size, the result blocks may come back at different time intervals asynchronously. In order to copy back the result block to $Matrix_C$, for each result block, the corresponding *blkid* is also returned. By checking the *blkid* value, the *master* program will be able to position the output at the right place. This procedure is demonstrated in Fig. C.2.

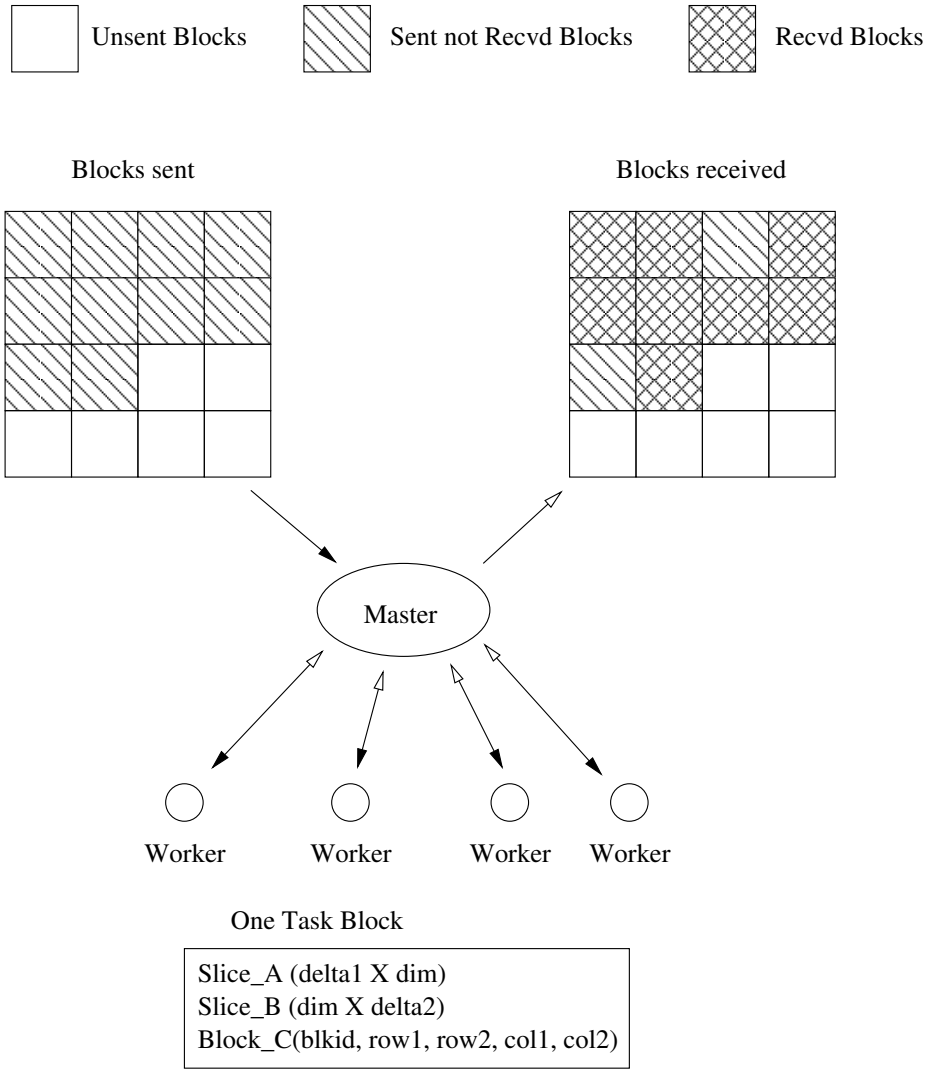


Figure C.2: Cooperation between the Master and Worker Programs

The *master* program keeps track of the number of blocks sent out and received. Upon receiving a result block, it will assign another block to the replier in case there are uncomputed blocks left. The whole computation procedure will stop once all result blocks have been received correctly. The

distributed algorithm to compute the multiplication product of two square matrix with the same dimension is given in Algorithm. 2.

Algorithm 2 Matrix Multiplication in Distributed mode

```
1: nblock - number of blocks to compute
2: nserver - number of workers
3: nsent - number of blocks sent to workers
4: nrecv - number of result blocks received from workers
5: for i = 0 to nserver do
6:    if nsent < nblock then
7:       send block[nsent] to server[i]
8:       nsent = nsent + 1
9:    else
10:      break;
11:   end if
12: end for
13: while nrecv < nblock do
14:   listen on all the nserver sockets
15:   if server[i] sends one completed block[t] back then
16:      copy the block[t] to result matrix
17:      nrecv = nrecv + 1
18:      if nsent < nblock then
19:         send block[nsent] to server[i]
20:         nsent = nsent + 1
21:      end if
22:   end if
23: end while
```