MONITORING NETWORK QUALITY OF SERVICE

IN A DYNAMIC REAL-TIME SYSTEM

A thesis presented to

the faculty of

the Fritz J. and Dolores H. Russ

College of Engineering and Technology of

Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Hong Chen

March 2003
This thesis entitled

MONITORING NETWORK QUALITY OF SERVICE
IN A DYNAMIC REAL-TIME SYSTEM

BY

HONG CHEN

has been approved for

the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

Carl Bruggeman
Assistant Professor of Electrical Engineering and Computer Science

Dennis Irwin
Dean, Russ College of Engineering and Technology
This thesis presents our design and implementation of a real-time network monitoring program for DeSiDeRaTa, an existing resource management system. This monitor will assist DeSiDeRaTa in maintaining an acceptable Quality of Service (QoS) for groups of real-time applications by reporting the communication delays caused by inadequate network bandwidth.

The network monitoring application we developed uses SNMP and network topology information gleaned from the DeSiDeRaTa application specification files. Algorithms were designed to compute network bandwidth utilization of each real-time communication path.

Preliminary experiments have been run on a local area network environment. The results demonstrated the accuracy of these measurements and the correct implementation of the network monitoring program.

Approved: Carl Bruggeman

Assistant Professor of Electrical Engineering and Computer Science
Acknowledgments

Thanks to my Mom, Dad, and my wife Ki and the rest of my family for all your love and support.

Thanks to Dr. Carl Bruggeman, Dr. Lonnie Welch, Dr. Shawn Ostermann, Dr. Brett Tjaden, and Barbara Pfarr for all your advisory and guidance.

Thanks to Lu Tong and the members of SWARM group and the LIRTSS lab for all your friendship and help.

Finally, the author gratefully acknowledges that this research project is funded by NASA Office of Earth Science, NASA Research Announcement 99-OES-08.
Abstract ........................................................................................................................................ 3
Acknowledgments ......................................................................................................................... 4
Table of Contents .......................................................................................................................... 5
List of Figures .................................................................................................................................. 8
List of Tables .................................................................................................................................... 10
Definition of Symbols and Abbreviations ...................................................................................... 11
Chapter 1 Introduction .................................................................................................................. 12
Chapter 2 Background .................................................................................................................. 15
  2.1 Resource Management and DeSiDeRaTa ............................................................................. 15
  2.2 Network Monitoring .............................................................................................................. 16
  2.3 SNMP ................................................................................................................................... 18
  2.4 Network Topology ................................................................................................................ 19
Chapter 3 Algorithms .................................................................................................................... 21
  3.1 Overview ............................................................................................................................... 21
  3.2 Network Topology and Path Traversal .............................................................................. 24
  3.3 SNMP Polling ....................................................................................................................... 26
  3.4 Bandwidth Calculation .......................................................................................................... 28
  3.5 Network Load Generator and Network Monitoring Display ............................................. 32
  3.6 UML Model of Network Monitor ....................................................................................... 36
Chapter 4  Implementation ........................................................................................................ 39
  4.1  Overview .......................................................................................................................... 39
  4.2  Network Topology ............................................................................................................ 40
    4.2.1  Specification ................................................................................................................ 40
    4.2.2  Data Structures .......................................................................................................... 40
    4.2.3  Interfaces .................................................................................................................... 42
  4.3  SNMP++ ............................................................................................................................ 43
    4.3.1  Overview .................................................................................................................... 43
    4.3.2  Usage of SNMP++ ...................................................................................................... 44
    4.3.3  SNMP++ API Functions .............................................................................................. 46
  4.4  Network Monitor .............................................................................................................. 50
    4.4.1  Overview .................................................................................................................... 50
    4.4.2  Multithreads and Mutex ............................................................................................ 52
    4.4.3  Communicating with Client Program .......................................................................... 54
  4.5  Accessory Functions and Programs .................................................................................. 55
    4.5.1  Time Functions .......................................................................................................... 55
    4.5.2  Network Load Generator ........................................................................................... 57
    4.5.3  Network Monitoring Display ...................................................................................... 58
    4.5.4  Programs Using SNMP++ API Functions .................................................................. 61

Chapter 5  Experiments and Results .......................................................................................... 63
  5.1  Chapter Outline ................................................................................................................ 63
  5.2  Experiment Setups ............................................................................................................. 63
List of Figures

Figure 3-1 Architecture of Network Monitoring ................................................................. 21
Figure 3-2 Overview of the Network Monitoring Program ................................................... 23
Figure 3-3 Schematic Diagram of Network Connections .................................................... 24
Figure 3-4 Specification of Network Topology ..................................................................... 26
Figure 3-5 Example of a Network Communication Path ..................................................... 30
Figure 3-6 Examples of Switch and Hub Connected Networks .......................................... 32
Figure 3-7 Algorithm of the Network Load Generator ....................................................... 34
Figure 3-8 Algorithm of the Network Monitoring Display ................................................. 35
Figure 3-9 Resource Instrumentation and Control Analysis Classes Diagram .................. 37
Figure 3-10 Network Monitor Design Sequence Diagram ................................................. 38
Figure 3-11 Network Monitor Design Class Diagram ......................................................... 38
Figure 4-1 Network Topology Data Structures .................................................................... 41
Figure 4-2 SNMP++ Framework ......................................................................................... 44
Figure 4-3 SNMP++ Example: Getting Single MIB Variable .............................................. 45
Figure 4-4 SnmpppApi Programming Example ................................................................... 50
Figure 4-5 Multithread and Mutex .................................................................................... 53
Figure 4-6 Data Structure of NmonMessage ....................................................................... 55
Figure 4-7 Example of Usage of TimeFcnt Class ............................................................... 57
Figure 4-8 Screenshot of the Network Monitoring Display ................................................. 59
Figure 5-1 Experiment Setups of LAN Test Bed ................................................................. 64
Figure 5-2 Experiment Results for Network Interface .............................................................. 67
Figure 5-3 Percentage Error of Measured Traffic ................................................................. 68
Figure 5-4 Average Packet Size ......................................................................................... 69
Figure 5-5 Overloading Switch Connected Network Link .................................................... 70
Figure 5-6 Revised Results for Overloaded Network Link .................................................... 71
Figure 5-7 Maximizing Hub Connected Network Link .......................................................... 72
Figure 5-8 Specification of Network Topology ..................................................................... 74
Figure 5-9 Experiment Results for Communication Path ...................................................... 75
Figure 5-10 Experiment Results for Hub-connected Network ............................................. 77
Figure 5-11 Experiment Results for Switch-connected Network ......................................... 79
List of Tables

Table 3-1 SNMP MIB-II Objects Used in Network Monitoring ........................................ 27
Table 4-1 Data and Functions of Network Topology Interfaces ........................................ 42
Table 4-2 Methods of SNMP++ API Class: SnmpppApi ...................................................... 47
Table 4-3 Java Classes Used in the Network Monitoring Display Program ...................... 60
Table 4-4 Programs Using SNMP++ API Functions ......................................................... 61
Table 5-1 Experiment Results for Network Interface ......................................................... 66
Definition of Symbols and Abbreviations

CORBA  Common Object Request Broker Architecture
GUI    Graphical User Interface
HCI    Human-Computer Interaction
IDL    Interface Definition Language
LAN    Local Area Network
MIB    Management Information Base
MTU    Maximum Transmission Unit
QoS    Quality of Service
RM     Resource Management or Resource Manager
SNMP   Simple Network Management Protocol
UML    Unified Modeling Language
Chapter 1 Introduction

Significant work has been done on resource management (RM) in distributed real-time computer systems. Such systems usually contain a large number of computers connected by one or more computer networks and many real-time applications. The quality of service (QoS) of these applications is critical for the performance of the whole computer system. Maintaining adequate QoS for real-time applications requires the resource management middleware to manage computer resources, including both computing resources and network resources. Computing resources are the basis for the execution of applications, e.g., computers, CPU time, memory etc., all of which directly affect the QoS an application can provide, while network resources provide means for individual computers to communicate with each other, e.g., network connections, interfaces, bandwidths etc.

In many cases, proper management of network resources is vital for delivering adequate QoS to a dynamic real-time system. A distributed real-time system usually consists of many real-time applications collaborating with each other. This may require that a large amount of information be exchanged between the various computers, resulting in a large amount of data communication over the network. An improper allocation of the applications or an improper selection of a network path can lead to congestion and delay of data transmission, and ultimately poor QoS. To properly manage network resources, the resource management middleware must be able to monitor their performance.
The research in this thesis concentrates on the techniques of network resource monitoring. It extends the work of DeSiDeRaTa, an existing resource management middleware for dynamic, scalable, dependable, real-time systems [1]. The current implementation of DeSiDeRaTa manages computer resources based only on the QoS of computing resources, assuming that network resources are never a bottleneck. Our work on network monitoring serves as an extension to DeSiDeRaTa so that the QoS of network resources can be taken into consideration when making resource management decisions.

In this thesis, techniques were designed to monitor the network resources for real-time systems. Software, which implements the techniques, has been developed and tested using the DeSiDeRaTa middleware. Our network monitoring program can provide real-time network performance information to the resource management middleware and help the middleware detect potential QoS violations due to a sub-optimal allocation of network resources.

Our approach is to combine the network topology and real-time data communication information to calculate the network performance information. We use the DeSiDeRaTa specification language to obtain the network topology. The network traffic information is gathered using SNMP (Simple Network Management Protocol) polling from computer hosts and network devices that make up the real-time system.

To obtain network performance information, the network resource monitoring program must know the topology of the computer network. The DeSiDeRaTa system includes a specification language with which to specify the components of a real-time system. For our work, we have extended the specification language to include network-
related information of the real-time system, such as computer hosts, network devices, network interfaces, and network connections. Our network monitoring program obtains the topology and connectivity of the real-time system from this specification file. An algorithm was implemented to traverse the communication path between hosts based on the topology information in the specification file.

The real-time data communication information for each network connection is obtained by periodically querying network components, including computer hosts and network devices, using SNMP. After combining the SNMP query results for each individual connection and network topology information, the bandwidth statistics (both available and used) between pairs of hosts is calculated. This bandwidth information can be used by the RM middleware to detect QoS violation and make proper decision on resource allocation.

The outline of the remainder of this thesis is as follows. Chapter 2 presents some background on this research, including related work on network monitoring, SNMP techniques, and the DeSiDeRaTa resource manager. Chapter 3 discusses in detail the algorithms of our approach of network monitoring, followed by Chapter 4, which gives the design and implementation of our programs. Chapter 5 provides some preliminary test results for the network monitor, along with some discussions. Chapter 6 presents the conclusions and gives some thoughts on future work.
Chapter 2  Background

2.1 Resource Management and DeSiDeRaTa

The work in this thesis is part of the effort to build DeSiDeRaTa, an adaptive resource management middleware for Dynamic, Scalable, Dependable Real-Time systems [1]. Such systems usually consist of computing resources, including computers and networks, and many real-time applications collaborating with each other. In addition, the applications have Quality of Service (QoS) requirements – they must meet their specified real-time deadlines. Sub-optimal allocation of resources might cause a computer to become overloaded or a network to become congested, and subsequently lead to a QoS violation. Therefore, resource management middleware becomes helpful, and often necessary, to manage the computer resources for real-time systems. The DeSiDeRaTa resource manager performs QoS monitoring and failure detection, QoS diagnosis, and reallocation of resources to adapt the system to maintain acceptable levels of QoS.

Prior to our work, DeSiDeRaTa managed only computational and not network resources. That meant that real-time systems controlled by DeSiDeRaTa could experience QoS violations caused by network delays and congestions. Our work extends DeSiDeRaTa to allow the management of network resources. We developed an application that does network QoS monitoring and provides the middleware with network metrics regarding data communication information, which enables the middleware to
manage the network resources based on the network metrics and network QoS specification.

2.2 Network Monitoring

Network monitoring is an important part of network resource management. The fast growth of computer networks and the Internet makes network monitoring more complicated and more challenging. Many different network monitoring techniques have been developed.

SNMP is a traditional technology and it has been widely used for network monitoring. Mansfield et al [2] studied a practical network management system using SNMP in the context of a large-scale campus network. Intelligent agents are deployed to collect information about network segments, and such information is reported to expert network manager. The network manager uses the information, in conjunction with a network knowledge base and a management information knowledge base, to reconstruct the overall network-traffic characteristic, to evaluate the status of the network, and to take or suggest some action. This technique uses time labels and narrow time windows to obtain a reasonably accurate picture of a large-scale network status. But it is not real-time and, therefore, cannot be used for our network monitoring.

Hedge et al [3] designed a tool, called netmon, for performance monitoring of a packet data network, which measured network performance statistics using SNMP information polled from the backbone routers. But this approach does not provide information of computer hosts and network path, hence, it is not suitable to be used as network monitoring tool for the resource management middleware.
A hierarchical network management solution, which also used SNMP, was proposed in [4] for large multi-layer networks. It provides a mid-level tool to structure network management hierarchically, and is dynamically configurable at runtime.

Many other techniques have also been developed for network monitoring. A Java- and CORBA-based network management prototype has been developed to manage distributed heterogeneous networks [5]. A distributed computing platform prototype was developed to support the creation, management, and invocation of distributed telecommunications services. It provides mechanisms that allow communication between CMIP-based (Common Management Information Protocol) objects, commonly used by industry, and a gateway for SNMP-based systems.


Another network monitoring tool that had been considered and actually tested in the course of this research was Remos (Resource Monitoring System) [16], which is currently being developed at Carnegie Mellon University. Remos allows network-aware applications to obtain both static and dynamic information about network [17]. The network uses SNMP and benchmark techniques to collect network status information.
However, Remos does not fit into our project well because one of its main efforts is to probe the whole network, which is unnecessary and sometime undesirable in our case. And furthermore, since it was still under development, the network status information Remos provided was unstable and incomplete according to our test results.

We considered these network monitoring techniques as the basis for our work. SNMP was chosen for the following reasons. First, SNMP provides very comprehensive network information. It not only contains all the information that fits our current needs, but also has much broader coverage that may be useful for future works. Second, SNMP is a well-developed technique. Many SNMP tools and interface programs are available, which makes it relatively easy for implementation. In addition, SNMP servers for different platforms (such as Solaris, Linux and Microsoft Windows) are available.

2.3 SNMP

The Simple Network Management Protocol (SNMP) provides a basic network-management tool for TCP/IP-based environments. Detailed descriptions of the protocol can found in RFC 1157 [9]. Although SNMP has been widely used for network monitoring, it is still under development to accommodate the quickly evolving computer networks. New versions of SNMP have been proposed for community-based networks (RFC 1901 [10]) and user-based security model (RFC 2574 [11]).

SNMP defines the structure of a hierarchically organized database, which is stored in a series of network components. Each network component contains information such as services offered, the device’s routing table, and a wide variety of statistics. Such information can be accessed through a local Management Information Base (MIB) [12]
that specifies the structure of the database. More information about SNMP and MIB can be found in [13].

Our network monitor uses the SNMP protocol and the interface table of the MIB to obtain data communication statistics from each SNMP-enabled network component. The interface table provides a static data transmission rate as well as counters for data and packets being sent and received through each network interface. Real-time network information is obtained by polling SNMP servers on each device (e.g. hosts, switches, routers) periodically. This data is then used to determine the amount of bandwidth used and available for each host and network device. By combining these metrics, we compute the available and used bandwidth of a real-time communication path.

2.4 Network Topology

SNMP gives network performance information for the network components. To obtain the network metrics, the network monitoring software has to be able to discover the network topologies and combine them with the SNMP data. Network topology discovery usually is difficult due to the increasing complexity of interconnectivity in computer networks.

We considered several different approaches for determining the topology of a network. Pure network discovery uses network protocols to discover hosts, routers, switches etc. and their connections. It is dynamic and powerful, and capable of discovering the whole Internet. However, the approach is too general and too sophisticated, which is beyond the scope of this research. Instead, we chose to utilize the hardware specification files that already exist for DeSiDeRaTa. The network topology
information is obtained from these files, which contain information about the network components and their interconnectivity. Using the specification files makes network topology discovery unnecessary, but results in a static topology, which must be known in advance. A hybrid approach is possible which would use specification files to obtain the initial network topology, and use network protocols to discover changes. For our work, we have chosen the approach that uses the static specification files. The DeSiDeRaTa middleware already has the infrastructure in place for this. A specification language has been developed to specify the information of the hardware and software systems under the control of DeSiDeRaTa. The specification is necessary because the middleware must know the details about hardware systems and all the software applications to be able to perform QoS management. For our work, the specification language was extended to include network components and connections. Our network monitor obtains this network information from the DeSiDeRaTa specification files and constructs the network topology graph for the system.
Chapter 3  Algorithms

3.1 Overview

The network monitoring program we have developed monitors network performance using network topology and data communication information from hosts and network devices. Figure 3-1 shows the architecture of our design for the network monitoring program. Network topology information was obtained from network specification files. Real-time data communication information of individual host or network devices is acquired by periodic SNMP polling. Our network monitor gathers all this information and calculates the real-time network performance metrics (bandwidth used and bandwidth available) for any pair of hosts in the system. Upon the request by the DeSiDeRaTa resource manager (RM), the network performance metrics are reported to assist the RM in managing the network resources.

Figure 3-1 Architecture of Network Monitoring
Figure 3-2 shows an overview of the network monitoring process. The program starts with loading and parsing DeSiDeRaTa specifications files. From these specification files, a network topology graph of the real-time system, as well as a list of computer hosts and network devices is created. The network monitor then makes connections to the SNMP servers on the hosts and network devices and initializes the SNMP communication.

The network monitor periodically makes SNMP queries and recalculates the network performance metrics. The program first sends an SNMP query to each SNMP server to gather the real-time data communication information of the hosts and devices. Based on this information and the network topology graph created at the beginning of the process, the amount of bandwidth between pairs of hosts is then calculated and stored so that they can be sent to the RM when requested.

Detail about the design and algorithm of network topology discover and path traversal is given in the next section. Section 3.3 describes the process of SNMP polling. Section 3.4 talks about the mathematic models that we used for bandwidth calculation. Section 3.5, describes two auxiliary programs we created to augment the network monitor – a network load generator and a network monitoring display. Section 3.6 gives the UML model of the network monitor in the resource management middleware.
Figure 3-2 Overview of the Network Monitoring Program
### 3.2 Network Topology and Path Traversal

The network-monitoring program was built using a local area network (LAN) model. The topology of such networks is modeled using hosts (or network devices), network interfaces, and network connections.

Figure 3-3 shows the LAN topology model. Each host or network device has one or more network interfaces. For example, in the figure, hosts A, C, and E each have a single connection (interface) to the network, while B and D have multiple interfaces. B and D can be hosts with multiple network connections, or network devices such as switches or hubs. A network connection is specified as a pair of interfaces that are physically connected to each other. In this model, the connection must be 1-to-1, *i.e.*, one interface may only be connected to one interface on another host/device.

![Network Connection Diagram](image)

**Figure 3-3 Schematic Diagram of Network Connections**

As mentioned earlier, we utilize the DeSiDeRaTa specification language to obtain this network topology information. A new extension to the DeSiDeRaTa specification
language [14] was developed to describe the information related to the network resources of the real-time system that is under control of the RM. The network topology is defined using the model described above.

Figure 3-4 shows the DeSiDeRaTa specification language for the definitions of data structures that store the network topology. A host/device is specified by its name, a list of all network interfaces on the host, and other host information. The interfaces are distinguished by their unique local names. A network connection is specified as two host-interface pairs, which give the two ends of the connection. Finally, the network topology can be described as a list of all the hosts and network devices and all the network connections among them.

The specification file is parsed, and related network topology information is stored using the data structures given in Figure 3-4. This results in a complete network topology graph that the network monitor can use. The communication path between any two hosts in the system can be traversed using a simple graph traversal algorithm. Given two hosts in the system, a simple recursive algorithm is designed to traverse the path, with a necessary infinite-loop detecting function implemented. The resulting communication path is described as a series of network connections defined in the previous section, starting with a connection from the source host, going through each hop in the communication path, and ending with a connection to the destination host.
3.3 SNMP Polling

The real-time network performance information is polled through SNMP from the SNMP-enabled hosts or network devices (i.e., the hosts/devices that have an SNMP demon running). There are hundreds of MIB-II (second version of MIB) objects in SNMP, which can provide abundant network information. Table 3-1 shows the particular ones that are requested by our network monitoring program.

Our interests are mainly focused on several interface entries in the network interface table (the objects prefixed with “interfaces.ifTable.ifEntry”, or “1.3.6.1.2.1.2.1.2.1” in numbers). The data transmitted through an interface in both directions, as well as the static bandwidth, can be obtained using these MIB-II objects.
Because the polling results are cumulative numbers, this data must be polled periodically. The old value is subtracted from the new one to determine statistics for the polling interval. The time interval between two polling processes can be found using the system uptime data. The data transmission rate, including packets and bytes per unit time can then be calculated.

### Table 3-1 SNMP MIB-II Objects Used in Network Monitoring

<table>
<thead>
<tr>
<th>MIB-II Object (Numbers)</th>
<th>Description [12]</th>
</tr>
</thead>
<tbody>
<tr>
<td>system.sysUpTime (1.3.6.1.2.1.1.3)</td>
<td>The time (in hundredths of a second) since the network management portion of the system was last re-initialized.</td>
</tr>
<tr>
<td>interfaces.ifTable.ifEntry.ifSpeed (1.3.6.1.2.1.2.2.1.5)</td>
<td>An estimate of the interface's current bandwidth in bits per second (static bandwidth).</td>
</tr>
<tr>
<td>interfaces.ifTable.ifEntry.ifInOctets (1.3.6.1.2.1.2.2.1.10)</td>
<td>Accumulated number of octets received on the interface.</td>
</tr>
<tr>
<td>interfaces.ifTable.ifEntry.ifInUcastPkts (1.3.6.1.2.1.2.2.1.11)</td>
<td>Accumulated number of subnetwork-unicast packets delivered to a higher-layer protocol.</td>
</tr>
<tr>
<td>interfaces.ifTable.ifEntry.ifOutOctets (1.3.6.1.2.1.2.2.1.16)</td>
<td>Accumulated number of octets transmitted out of the interface.</td>
</tr>
<tr>
<td>interfaces.ifTable.ifEntry.ifOutUcastPkts (1.3.6.1.2.1.2.2.1.17)</td>
<td>The total number of packets that higher-level protocols requested to be transmitted to a subnetwork-unicast address.</td>
</tr>
</tbody>
</table>

For example, assume the SNMP results of polling number \( i \) are \( t(i) \) for system uptime and \( b_{in}(i) \) for received data bytes (octets). The results of previous polling were \( t(i-\)
1) and $b_{in}(i-1)$ correspondingly. Then the data bytes received by the host during the period is:

$$\Delta b_{in}(i) = b_{in}(i) - b_{in}(i-1).$$  \hfill (3-1)

And the actual time interval between SNMP polling can be given by:

$$\Delta t(i) = t(i) - t(i-1).$$  \hfill (3-2)

Notice that $\Delta t(i)$ may be slightly different (normally one to two hundredths of a second) with the interval between the moments when SNMP requests are sent (normally 1 second in our case). This is likely caused by the delay in the network communication and response time of SNMP servers. $\Delta t(i)$ gives a more accurate measurement, and therefore it is used for the transmission rate calculations. The average incoming data transmission rate for period $i$ can be expressed as:

$$r_{in}(i) = \frac{\Delta b_{in}(i)}{\Delta t(i)} = \frac{b_{in}(i) - b_{in}(i-1)}{t(i) - t(i-1)}.  \hfill (3-3)$$

Similarly, the outgoing data transmission rate and the rates for both incoming and outgoing packets can be calculated.

### 3.4 Bandwidth Calculation

The specification files and SNMP polling provide the network monitor with network topology and data communication information of individual network connections in the system, respectively. Based on such information, the network monitoring program should be able to calculate the bandwidth information of a communication path between two given hosts, which consists of a series of connections.
The available bandwidth of a communication path is simply the minimum of the available bandwidths of all the network connections along the path. Assume a communication path consists of \( n \) network connections, and the available bandwidth for connection \( i \) is \( a_i \) \((i = 1, 2, \ldots n)\). Then the available bandwidth of the path, \( B_{\text{available}} \), can be given by the following equation:

\[
B_{\text{available}} = \min(a_1, a_2, \ldots a_n) \tag{3-4}
\]

The maximum bandwidth of the communication path, \( B_{\text{max}} \), is defined as the minimum value of the maximum bandwidth \((m_i)\) of each individual connection along the path:

\[
B_{\text{max}} = \min(m_1, m_2, \ldots m_n) \tag{3-5}
\]

The used bandwidth of the communication path, \( B_{\text{used}} \), can then be given as:

\[
B_{\text{used}} = B_{\text{max}} - B_{\text{available}} \tag{3-6}
\]

Figure 3-5 gives an example of a network communication path from host A to host D, which in order consists of three network connections, 1, 2, and 3. The computed maximum, utilized and available bandwidths for each connection are \( m_i, u_i \) and \( a_i \) \((i = 1, 2, 3)\), respectively. Then the maximum bandwidth of the communication path is given by \( B_{\text{max}}(A-D) = \min(m_1, m_2, m_3) = \min(100, 100, 10) = 10 \text{ Mbps} \). The available bandwidth of the path is \( B_{\text{available}}(A-D) = \min(a_1, a_2, a_3) = \min(80, 15, 5) = 5 \text{ Mbps} \). Both bandwidths are decided by the bottleneck in the communication path, connection 3 between C and D. Finally, the total utilized bandwidth between A and D is \( B_{\text{used}}(A-D) = B_{\text{max}}(A-D) - B_{\text{available}}(A-D) = 10 - 5 = 5 \text{ Mbps} \).
For each individual connection, the available bandwidth $a_i$ is just the difference between maximum bandwidth $m_i$ and used bandwidth $u_i$:

$$a_i = m_i - u_i.$$  \hspace{1cm} (3-7)

In Figure 3-5, $a_1 = m_1 - u_1 = 100 - 20 = 80$ Mbps. A measure of $m_i$ can be obtained directly through SNMP polling, while the $u_i$ must be computed by the network monitor.

It is relatively easy to calculate used bandwidth of a host connected to switches because a switch does not forward packets for one host to other hosts connected to the same switch. Hence, the amount of bandwidth used on a host connected to a switch is simply the amount of data transmitted as reported by SNMP polling from either the host or the switch. If the traffic reported is $r_i$, then we simply have

$$u_i = r_i.$$ \hspace{1cm} (3-8)

If machines A, B, and C are all connected to the same switch, and each is sending 3 Mbps to a fourth machine, D, the amount of bandwidth used for each is: $u_A = u_B = u_C = 3$ Mbps. The traffic to another connection $r_j (j \neq i)$ will not affect $u_i$.

However, for hosts connected to hubs, all packets that go through the hub will be sent to every host connected to the hub. Therefore, the amount of bandwidth used for a host connected to a hub is the sum of all the data sent to the hub, instead of just the one to
this host. Assume there are \( n \) hosts connected to the hub and the traffic reported by SNMP polling for host \( i \) is \( r_i \) \((i = 1, 2, \ldots n)\). Then

\[
u_i = \sum_{i=1}^{n} r_i \tag{3-9}\]

If machines A, B, and C are all connected to the same hub, and each is sending 3 Mbps, the amount of bandwidth used for each is: \( u_A = u_B = u_C = 9 \) Mbps. \( u_i \) cannot exceed the maximum speed of the hub.

An algorithm was implemented in our network monitoring program to distinguish these two cases and calculate the amount of bandwidth used accordingly for each network connection along the communication path. Combining the used bandwidths and the maximum bandwidths for these connections, the maximum, available, and used bandwidth of the communication path between the two hosts can be calculated using equations 3-4 to 3-7.

Figure 3-6(a) shows an example of three hosts (A, B and C) connected through a switch. Network traffic is generated from A to B and C. The traffic loads, which are reported by the SNMP servers on B and C, are \( u_B \) and \( u_C \), respectively. Since the switch is capable of distinguishing which traffic is for which host, the traffic of \( u_B \) is only forwarded to host B. Therefore the used bandwidth of connection 2 between host B and the switch can be given by \( u_2 = u_B \). Similarly, the used bandwidth of connection 3 between host C and the switch is \( u_3 = u_C \).

Figure 3-6(b) shows a similar network structure but connected through a hub. The same traffic loads of \( u_B \) and \( u_C \) are generated. However, the hub forwards the traffic to
every hosts connected to the hub. In this example, the traffic of $u_B$ is sent not only to host B, but also to host C. Notice that although $u_B$ appears on connection 3 between host C and the hub, it will not be reported by the SNMP server on host C because it is not designated to host C. The same situation happens to the traffic $u_C$. The used bandwidths of both connection 2 and connection 3 are equal to the sum of $u_B$ and $u_C$, i.e. $u_2 = u_3 = u_B + u_C$.

![Diagram](image)

**Figure 3-6 Examples of Switch and Hub Connected Networks**
(a) a switch-connected network. (b) a hub-connected network.

### 3.5 Network Load Generator and Network Monitoring Display

Several auxiliary programs were built in this research to assist the development and testing of our network monitor. Among them are a network load generator, which generates a specified amount of network traffic flow to a designated host, and a network
monitoring display, which graphically displays the real-time bandwidth information provided by the network monitor between a pair of hosts.

The network load generator sends data packets at a constant rate from a local host to a designated remote host. Users are able to specify which host the data is sent to, and at what rate (Kbytes/second). Data is sent in a burst with a constant time interval between any two consecutive bursts. Users can also specify the number of bursts in each second. To generate a network traffic load of $N$ Kbytes/second in an $n$ bursts/second pace, the load generator will send a burst of $N/n$ Kbytes of data in every $1/n$ second. To limit the overhead on the receiving host and to avoid possible retransmission of data packet, the load generator program sends the data in UDP packets to the UDP Discard port (port number 9) of the remote host.

The algorithm of the network load generator is shown in Figure 3-7. The program first calculates the size of each data burst and the time interval between bursts based on user specifications. Then it connects to the UDP Discard port of the designated host. Upon successful connection, the load generator sends data bursts periodically through the connection until the process is stopped by the user.

The network monitoring display is a graphical user interface (GUI) program to show the real-time bandwidth information computed by the network monitoring program. It shows the current available bandwidth or used bandwidth between a pair of hosts monitored by the network monitor. The display program provides a more direct view of the results of our network monitor by showing the history curve of the bandwidth data. It
also serves as a prototype of the network monitoring part of the human-computer interaction (HCI) program of the DeSiDeRaTa middleware.

![Network Load Generator Diagram](image)

**Figure 3-7 Algorithm of the Network Load Generator**

Figure 3-8 gives the algorithm of the network monitoring display program. The program runs as a server, which communicates with the network monitor through TCP/IP, so that the two programs can be run at different hosts. The display first opens a socket at a dynamically selected port number and waits for the data from network monitor. The network monitor makes the connection and then periodically sends the...
observed bandwidth information of a communication path between a pair of hosts, together with the name of these two hosts. After the connection from the network monitor, the display program starts two threads, one receives the data from the network monitor and update the bandwidth information, while the other thread periodically refresh the GUI display with the updated data.

Figure 3-8 Algorithm of the Network Monitoring Display
3.6 UML Model of Network Monitor

The Unified Modeling Language (UML) is a standard language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system [18]. UML can be used to write software blueprints with object-oriented design. As part of the effort of reengineering the resource management middleware using object-oriented design, the network monitoring program was modeled using UML.

Figure 3-9 shows the analysis of “Resource Instrumentation and Control” of the resource management middleware. The network monitor belongs to the “Resource Monitor Service Package” along with the host monitor, which monitors the resources of computer hosts. The network monitor and the host monitor share some similarities in the design. Each has an interface (network monitor interface or host monitor interface) to report the resource utilization information to the “Allocation Manager”, and each has another interface (network monitor network interface or host monitor host interface) to gather the information from the hosts and network devices. The monitoring results are stored in a “Resource” object.

Figure 3-10 shows the design sequence of the network monitor. The allocation manager sends request of network information to the network monitor through the network monitor interface. The network monitor gets a resource template to store the network information. The network information of each individual host or network device is gathered through the network monitor network interface. Such information is then processed by the network monitor and reported to the allocation manager.
Figure 3-11 shows the three design classes for the network monitoring program. The “NetworkMonitor” class is the main class of the network monitor. The “NetworkMonitorInterface” class provides a communication interface between the allocation manager and the network monitoring program. The communication uses CORBA (Common Object Request Broker Architecture), and the interface is specified using CORBA (IDL Interface Definition Language). The interface with the hosts and network devices is defined as “NetworkMonitorNetworkInterface” class. It connects the network monitor with the SNMP servers on the hosts and network devices.
The implementation of the UML model described in this section is being worked on during the writing of this thesis. Therefore the following chapters of this thesis are based on the early designs.
Chapter 4  Implementation

4.1 Overview

In this chapter, we present the implementation of the network monitoring software and some accessory programs. The programs were initially developed in a Linux environment mainly using the C programming language. Later they were migrated into the Unix environment and the C++ language. The implementation discussed in the following sections is based on the latest development of the programs. The usage of the programs, as well as the functionalities of some API functions we developed, are also presented in detail, which may be used as the documentation or user manual for these programs.

Section 4.2 gives details on the implementation related to network topology, including specification, data structures that store the topology information, and the program interfaces. Section 4.3 discusses a SNMP developing tool called SNMP++, which we used to build the interfaces for SNMP polling. It first briefly introduces the SNMP++, and then discusses the API functions that we built for our monitoring program. Section 4.4 focuses on the network monitoring program, giving details on integration and program usage. Finally, some accessory programs are presented in Section 4.5, including the time function, the network load generator, the network monitoring display, and some other utility programs.
4.2 Network Topology

4.2.1 Specification

A new extension to the DeSiDeRaTa specification language was developed to describe the information related to the network resources of the real-time system that is under control of the RM. The hosts, devices, interfaces, and network connections of the real-time system are included in the RM hardware specification files.

Our network monitoring program obtains the network topology information from the specification files. The next section explains the data structure being used in our program to store the topology information. Section 4.2.3 describes the program interface we built to acquire the network topology.

4.2.2 Data Structures

Figure 4-1 gives the three pseudo-data structures being used in our program to store the network topology information. The “nmon_host” holds information of an interface of on a host or network devices. It contains the host name and the interface name, and the hardware type (host, hub or switch). A unique “id” can be assigned by the SNMP++ API program (see Section 4.3.3). It also has space to store the bandwidth information of the interface. For a host or device having multiple network interfaces, one “nmon_host” is created for each interface.

The “nmon_host_pair” data structure holds two pointers to two network interfaces (“nmon_hosts”), which map a physical connection between two network cards on two machines. As has been explained earlier, for a connection, only one end needs to be
monitored to obtain the network information about the connection. Therefore, an “index” is used to mark which network interface is used for SNMP polling.

```c
/** A network host (host and interface) **/
typedef struct nmon_host {
    int id;        //host-interface id
    char *type;    //hardware type (host, switch, hub)
    char *name;    //host name
    char *hostif;  //interface name
    int maxBandwidth;  //maximum bandwidth (bytes/cycle)
    int inBandwidth;   //total incoming bandwidth (bytes/cycle)
    int outBandwidth;  //total outgoing bandwidth (bytes/cycle)
} nmon_host;

/** A pair of nmon_hosts that are physically connected **/
typedef struct nmon_host_pair {
    int index;    //index of host for snmp polling
    nmon_host *h[2];  //pointers to two nmon_host's
} nmon_host_pair;

/** A network device (hub or switch) **/
typedef struct nmon_dev {
    char *name;    //device name
    char *type;    //hardware type (switch, hub)
    int maxBandwidth;  //maximum bandwith (bytes/cycle)
    int inBandwidth;    //total incoming bandwidth (bytes/cycle)
    int outBandwidth;   //total incoming bandwidth (bytes/cycle)
    Linked_List *hosts;  //list of connected nmon_hosts
    Linked_List *hp;     //list of connected nmon_host_pairs
} nmon_dev;
```

**Figure 4-1 Network Topology Data Structures**

The “nmon_dev” data structure stores information of a network device. It gives the name and type of the device, a list of all “nmon_hosts” connected to the device, and a list of network connections (“nmon_host_pairs”) to the device. The bandwidth information may be computed based on the bandwidth of each connection, and different computation algorithm may be used depending on the type of the device (hub or switch).
4.2.3 Interfaces

For easy access to the network topology information, interfaces were built for our network monitor (Table 4-1). Three linked lists are used to store the network topology information. The “hlist” is a list of all the network interfaces, which may be polled by SNMP. The “hplist” maps all physically connection in the network. The “dlist” is a list of all network devices.

<table>
<thead>
<tr>
<th>Data and Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hlist</td>
<td>Linked list of “nmon_hosts” that stores all the interfaces of hosts and devices in the network.</td>
</tr>
<tr>
<td>hplist</td>
<td>Linked list of “nmon_host_pairs” that stores all the physical connection between hosts and devices.</td>
</tr>
<tr>
<td>dlist</td>
<td>Linked list of “nmon_devs” that stores all the network devices (hubs and switches).</td>
</tr>
<tr>
<td>void topo_init()</td>
<td>Initialize network topology from specification files. Initialize the lists of “hlist”, “hplist”, and “dlist”.</td>
</tr>
<tr>
<td>Linked_List * find_path(char *h1, char *h2)</td>
<td>Find communication path between two hosts. Arguments: names of two hosts. Return value: an ordered list of “nmon_host_pairs” mapping the physical path if found; NULL otherwise.</td>
</tr>
</tbody>
</table>

The function topo_init() obtains network information from network specification and initialized the three lists. Using the lists, the network monitoring program creates a list of network interface to be monitored. Bandwidth usage information of each network connection can be obtained through SNMP polling on the interfaces, and the usage of a network device can be computed.
The “find_path()” function finds communication path between two hosts. It uses the list of network connections, “dlist”, to traverse the path and returns the result as an ordered list of connections from host 1 to host 2, with each connection represents one hop in the physical network path between the hosts. For each network path being monitored, the communication path is first traversed using “find_path()” function. The bandwidth information of the path is then computed using the result for each connection and network device through the path.

4.3 SNMP++

4.3.1 Overview

SNMP++† is an open specification for object oriented network management development using SNMP and C++. It was developed by the Hewlett Packard Company. SNMP++ provides a C++ based application programmers interface for SNMP. We chose SNMP++ as the programming interface between our network monitoring program and SNMP because of its ease of use and portability.

SNMP++ hides the sophisticated detail of SNMP from user applications. It also uses highly Object Oriented approach, while the implementation is kept simple, and does not require any expertise on the C++. The result is any easy to use, straightforward API. The SNMP++ API has great portability across a variety of operating systems and network management platforms (Figure 4-2). It supports both UNIX and Windows SNMP while

hiding their differences from users. A user who codes to SNMP++ does not have to make changes to move the codes to another platform.

Figure 4-2 SNMP++ Framework

4.3.2 Usage of SNMP++

With the SNMP++ API, SNMP programming becomes simple and straightforward. In this section, we illustrate how to use the SNMP++ interface to do SNMP polling using a short example (Figure 4-3). The function in the example polls the SNMP server on a specified host for the system uptime of the server. A line-by-line breakdown below may help the readers to get familiar with SNMP++ interfaces.

First, to use SNMP++, the user must obtain the SNMP++ package and compile it. The first line of the sample code includes the SNMP++ header file “snmp_pp.h”, which resides in the “include/” sub-directory in the SNMP++ package. To compile the program,
two libraries must be linked: “lib/libsnmp++.a” and “libdes/libdes.a”. The next line gives the OID of the MIB variable being polled. The OID of a MIB variable can be looked up in RFC 1213 [12].

```c
#include "snmp_pp.h"
define SYSUPTIME "1.3.6.1.2.1.1.3.0" //system uptime object ID

void get_system_uptime(char *host) { //argument: host name
    int status;
    GenAddress genAddress(host); //SNMP++ generic address
    CTarget ctarget(genAddress); //SNMP++ target
    Vb vb(SYSUPTIME);
    Pdu pdu;

    Snmp snmp(status);
    if(status!=SNMP_CLASS_SUCCESS) {
        cout<<snmp.error_msg(status);
        return;
    }

    pdu+=vb; //add variable binding
    status=snmp.get(pdu,ctarget); //get data
    if(status!=SNMP_CLASS_SUCCESS)
        cout<<snmp.error_msg(status);
    else {
        pdu.get_vb(vb,0); //extract variable binding
        cout<<"System Uptime = "<<vb.get_printable_value();
    }
}
```

**Figure 4-3 SNMP++ Example: Getting Single MIB Variable**

The example function takes the string of host name as the only argument. A generic address “genAddress” is generated for the host, and the address is used to set up the SNMP polling target “ctarget”. Various SNMP polling options, such as version, timeout, can be set using different “CTarget” set methods. A variable binding object “vb” is
created for the system uptime MIB variable. The protocol data unit object “pdu” is used to carry the variable binding data.

Next, an SNMP session is created and the status is checked. The “pdu+=vb” statement adds the “vb” to the “pdu”. For multiple variables, an array of “Vb” objects can be created and added to the “Pdu” object one by one. The “Pdu” object is then sent to the target host using the “get()” method of the “Snmp” object to acquire the SNMP information. If the polling succeeds, the result is then printed out. In the example, the “vb.get_printable_value()” method is used for the output. The result can also be extracted using corresponding “get_value()” method.

With SNMP++, the SNMP programming becomes simple and straightforward. Moreover, the programs are portable over different platforms.

### 4.3.3 SNMP++ API Functions

To make the SNMP++ better serve our network monitoring program, several higher level API functions were built over SNMP++ interfaces. These functions were wrapped in a class called “SnmpppApi”. Table 4-2 shows a list of the major functions and their description. To use the “SnmpppApi” class, a program needs to include the class header file “SnmpppApi.h” and to be compiled with “SnmpppApi.cc”.

The “SnmpppApi” class has two constructors. The default constructor uses default SNMP settings, while the other one allows user to specify the setting. The first argument sets the SNMP version (version 1 or 2, 1 by default). The second argument sets the number of retries, which specifies how many SNMP calls will be made, if all failed, before an SNMP polling is aborted (default once). The third argument gives the timeout
for an SNMP call, *i.e.* an SNMP call is thought to be failed if the response is not received before the timeout. The default value is 100 (1 second). The last argument sets the community of SNMP.

<table>
<thead>
<tr>
<th>Class Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SnmpppApi()</td>
<td>Default constructor.</td>
</tr>
<tr>
<td>SnmpppApi(int, int, int, char*)</td>
<td>Constructor. User specifies SNMP settings, in the order of argument list: version, number of retries, timeout in hundredth of second, and community.</td>
</tr>
<tr>
<td>int add_host(char*, char*)</td>
<td>Add a host/network interface pair. Arguments: host name and interface name. Return value: a unique host/interface ID, or -1 if failed.</td>
</tr>
<tr>
<td>char** hostif(int)</td>
<td>Find names of all interfaces of a specified host. Argument: host/interface ID. Return value: list of interface names ending with NULL.</td>
</tr>
<tr>
<td>int snmp_data(int, SnmpData*)</td>
<td>Get SNMP data from a specified host. Arguments: host/interface ID and address of an SnmpData structure to store the returned SNMP data. Return value: 0 if call succeeded; -1 otherwise.</td>
</tr>
</tbody>
</table>

The “SnmpppApi” class keeps a list of hosts and their interfaces. A host/interface pair must be added to the list before SNMP data can be polled for the pair. The function “add_host()” takes two strings as host and interface name. If the host is new, it is added to the host list and a series of SNMP++ calls are made to initialize the entry:

- Get the generic address of the host;
- Set “CTarget” for the host and apply the SNMP settings;
- Make SNMP++ calls to get the number of network interfaces on the host;
• Make SNMP++ calls to get the indices of all the network interfaces;
• Make SNMP++ calls to get the names of all the network interfaces;
• Store the indices and name of all the interfaces in the host list.

The host/interface pair is looked up in the host list and a unique ID, which is a combination of host ID and interface index, is returned. The ID will be used for the following SNMP polling calls.

The “hostif()” function returns the list of names of network interfaces on a specified host. The host/interface ID is passed as argument, although only the host ID portion of the ID is used and the interface index part is ignored. To discover the interfaces, one can simply use the “SnmpppApi” class as following:

```cpp
SnmpppApi snmpppapi();
int id = snmpapi.add_host("agent","" Jewish");
char** ifnames = snmpapi.hostif(id);
```

The names of interfaces on host “agent” are stored in the array of strings “ifnames”. An empty string is passed in as interface name for “add_host()” call, and the output “id” contains a valid host ID and an invalid interface index. The “id” can be used for the “hostif()” call since only the host ID part is required. However, it can not be used in the “snmp_data()” function calls.

The method “snmp_data()” takes a valid host/interface ID and an empty “SnmpData” data structure as argument. Upon success, the “SnmpData” is filled with latest SNMP polling results. The “SnmpData” data structure contains the host and interface names as strings and all the SNMP data as unsigned long integer corresponding with the MIB objects listed in Table 3-1. The “snmp_data()” works as following:
- Retrieve host information in the host list using the host/interface ID;
- Create “Vb” objects for each MIB objects and add them to a “Pdu” object;
- Poll the target host using the host “CTarget” and the “Pdu” object;
- Extract polling results from “Pdu” and “Vb” objects and fill in the “SnmpData”.

Upon success, the return value is set to 0, otherwise, -1 is returned.

An example code is shown in Figure 4-4. The get_snmp_data() function gets SNMP data for a specified host/interface pair. The process is very simple using the “SnmpppApi” class. An instance of the class is first initiated, then the host/interface pair is added to the host list. Upon success, a valid ID is assigned for the pair. This ID, along with the address of a new “SnmpData” data structure are passed to “snmp_data()” for SNMP polling. Two data samples are polled with a time interval of about 1 second. The data transmission rates are calculated and printed out. There are a few things need to be pointed out. First, although “sleep(1)” is called between the two SNMP polling processes, the actual time interval for the two sets of data may not be 1 second. Instead, the system uptime field can give us an accurate measurement. The second point is the time interval “t” derived from the system uptimes are checked against zero before the transmission rates are calculated to avoid divided-by-zero errors. The reason is that, despite the 1-second interval between two queries, the same set of data sample may be returned in some cases depending on the behavior of the SNMP server. Some SNMP servers may update their data in a period that longer than 1 second.
```c
#include "SnmpppApi.h"

void get_snmp_data(char *host, char *nif) {
    //Initialize SnmpppApi and host/interface pair
    SnmpppApi snmpppapi(); //new SnmpppApi object
    int id=snmpppapi.add_host(host,nif); //add host/interface
    if(id<0) return; //invalid host/interface

    //Make SNMP polling
    SnmpData data0, data1; //for storing results
    int i=snmpppapi.snmp_data(id,&data0); //SNMP polling
    if(i!=0) return; //error occurred
    sleep(1); //sleep 1 second
    int i=snmpppapi.snmp_data(id,&data1); //SNMP polling
    if(i!=0) return; //error occurred

    //Calculate and print out polling results
    int t = data1.sysUpTime - data0.sysUpTime //time interval
    if(t==0) return; //data unusable
    int r_in = (data1.ifInOctect-data1.ifInOctect)*100/t;
    int r_out = (data1.ifOutOctect-data1.ifOutOctect)*100/t;
    cout << "Rate(in) = " << r_in << " bytes/sec" << endl
         << "Rate(out) = " << r_out << " bytes/sec" << endl;
}
```

**Figure 4-4 SnmpppApi Programming Example**

### 4.4 Network Monitor

#### 4.4.1 Overview

We built the network monitoring program using C++ and SNMP++ tools. The program we built is named as “nmon”. The usage of “nmon” is:

```
nmon [Nmon_options] [Snmp_options]
```

There are two “Nmon_options” can be specified by user:

“-CN”: where N is the cycle of SNMP polling in hundredth-seconds, with default N=100 (1 second); and
“-Dhost:port”: which specifies the host and port number of network monitoring display. Multiple displays can be given using multiple “-D” options. For example, command “nmon -C200 -DN1:2000 -DN1:2001” starts the program with a cycle of 2 seconds and specifies two displays on machine “N1”, port 2000 and 2001. The “Snmp_options” specify the SNMP settings:

“-v[12]”: using SNMP version 1 (default) or SNMP version 2;

“-cCommunity”: specifying SNMP community, with default value of ‘public’;

“-rN”: number of retries for SNMP polling, with default N=1 retry; and

“-tN”: timeout for SNMP calls in hundredth-second (default N=100, 1 second).

The “Snmp_options” have been discussed in Section 4.3.3.

When the “nmon” is started, it runs as following:

- Parsing the command line options;
- Initializing network topology information from network specifications;
- Creating an “SnmpppApi” object and adding network hosts and network devices for SNMP polling;
- Starting two parallel processes (threads):
  - One process opens a listening port and waits for requests from the client program. The client sends a request by giving two names of the end hosts of a communication path. If the host pair is new, the path will be found and added to the list of paths. If the path already exists, the latest results of bandwidth will be sent to the client.
The other process does SNMP polling periodically and gathering real-time network information from hosts and devices. The bandwidth information of each path in the path list is computed and updated.

Details about multithread programming and communication with the client program are given in the following sections.

4.4.2 Multithreads and Mutex

Multithreading was used to perform the two parallel jobs: monitoring network performance and answering request from resource manager. Pthread, a POSIX standard thread interface, was used in our program.

Two threads are created, one for each process. The most recent network metrics are stored in a section of shared memory. The thread for network monitoring updates the memory periodically, while the other thread reads from the memory and sends the metrics to the resource manager per request. To avoid conflicts during the access of the shared memory, mutex, a mutual exclusion lock provided by pthread, is used.

Figure 4-5 shows the part of the code involving the pthread programming. At the beginning, a mutex and two thread functions are declared. The mutex is initialized in the “main()” using “pthread_mutex_init()” function, and it is destroyed after the task is finished using “pthread_mutex_destroy()”. The mutex is used in the two thread functions to protect the critical sections.

Two pthreads are created using “pthread_create()” calls. The third and the fourth arguments of the function is the name of thread function and its argument, respectively. This will start two threads, which execute the two functions, “thread_snmp()” and
“thread_nmd()”, at the same time. By using “pthread_join()” calls, the main process is suspended until both threads complete.

```c
#include <pthread.h>  // header file for pthread and mutex

pthread_mutex_t MUTEX;  // declare mutex

void* thread_snmp(void* p);  // thread to gather network metrics
void* thread_nmd(void* p);  // thread to communicate with RM

int main() {
    ......
    pthread_mutex_init(&MUTEX,NULL);  // initialize mutex
    pthread_t t1, t2;
    pthread_create(&t1,NULL,thread_snmp,(void*)pdata);
    pthread_create(&t2,NULL,thread_nmd,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    pthread_mutex_destroy(&MUTEX);  // destroy mutex
    ......
}

void* thread_snmp(void* p) {
    ......
    pthread_mutex_lock(&MUTEX);
    ....../critical section: update network metrics data
    pthread_mutex_unlock(&MUTEX);
    ......
}

void* thread_nmd(void* p) {
    ......
    pthread_mutex_lock(&MUTEX);
    ....../critical section: read network metrics data
    pthread_mutex_unlock(&MUTEX);
    ......
}
```

**Figure 4-5 Multithread and Mutex**

The “thread_snmp()” function periodically polls the SNMP servers on the network hosts and devices, and computes the bandwidth usage of given communication paths. To update the bandwidth stored in the shared memory, the mutex is locked before updating
the data using “pthread_mutex_lock()”. The mutex is unlocked after writing is finished using “pthread_mutex_unlock()”.

The “thread_nmd()” is more like a network monitoring daemon, which answers request from client program and reports bandwidth usage of communication paths. Whenever requested, the latest data is obtained from the shared memory and returned to the client. The reading of shared memory is protected by the mutex.

4.4.3 Communicating with Client Program

The communication between the network monitor and client program uses TCP/IP. The data structure used as both request and response are shown in Figure 4-6. There are two strings of host names, which need to be filled by the client program and sent to the network monitor. Remaining fields, available and used bandwidth, and returning status, will be filled by the network monitor.

When the return status is “NMMSG_OK”, the request is successful and the bandwidth information of the path is stored in the fields “available” and “used”. Otherwise, no bandwidth information is returned. “NMMSG_NOPATH” means no communication path can be found between the given hosts. This may be caused by the wrong host name or incomplete network specification. “NMMSG_NEWPATH” means the path between the given hosts is new and has just been added to the list of network monitoring program. Bandwidth information will be available at least one cycle (typically 1 second) later. “NMMSG_WAIT” means the network monitor is still waiting for SNMP polling results to compute the bandwidth.
#define NMMSG_MAX_NAME_LEN 64
#define NMMSG_OK 0  // returned value okay
#define NMMSG_NOPATH 1  // no path found between the host pair
#define NMMSG_NEWPATH 2  // new path is added for the host pair
#define NMMSG_WAIT 3  // waiting for network metrics data

struct NmonMessage {
    char host1[NMMSG_MAX_NAME_LEN];  // host name #1
    char host2[NMMSG_MAX_NAME_LEN];  // host name #2
    int available;  // available bandwidth
    int used;  // used bandwidth
    int status;  // return status
    int padding;
};

Figure 4-6 Data Structure of NmonMessage

When the network monitoring program starts, it opens a listening port and waits for the connection from the client program. The client program connected to the machine on which the network monitor is running at the listening port number through TCP/IP. The request is sent to the monitor by specifying the two end hosts of the path. If the path is new for the network monitor, the path will be retrieved and stored in the list of paths being monitored. If the path is already in the list, the latest bandwidth information will be returned to the client program.

4.5 Accessory Functions and Programs

4.5.1 Time Functions

The programming in this project involves many periodic tasks such as SNMP polling, network load generating. The most straightforward approach is to use the C library function “usleep()”. For example, to run a task in period of 1 second, simply call `usleep(1000000)` between each cycle. The approach is not accurate since the execution
of task in each cycle may cost time. The error can be significant for tasks, such as SNMP polling, which may take long time.

A modified approach is to obtain the time stamps before and after the task in each period using the C library function “gettimeofday()”. The actual time used on the task can then be calculated and the time to “sleep” is the period subtracted by the time used. This can give a much accurate results than the previous approach. The actual time is about 1.01 second for the cycle of 1 second, i.e. 1% longer, due to the delays in the “gettimeofday()” and “usleep()” function calls. Such error is cumulative and can become significant when the program runs many periods.

We did further modification by adding an error control mechanism. Besides measuring the actual task execution time, the actual time of one cycle is measured. Let \( T \) be the required cycle time, \( T_{\text{cycle}}(i) \) and \( T_{\text{task}}(i) \) be the actual cycle time and task execution time for cycle number \( i \). Let \( \text{Error}(i) \) be the error correction for cycle number \( i \) and \( \text{Sleep}(i) \) be the time used for “usleep()” call in cycle \( i \). Then \( \text{Sleep}(i) \) is calculated as follows:

\[
\text{Sleep}(i) = T - T_{\text{task}}(i) - \text{Error}(i) \tag{4-1}
\]

\[
\text{Error}(i) = \text{Error}(i-1) + T_{\text{cycle}}(i-1) - T \tag{4-2}
\]

The initial value of \( \text{Error}(0) \) is 0. Using this technique, the system error caused by the C library function calls is eliminated, and the long-term cumulative error becomes 0, although there is still random error during each period.

For easy use of this technique, we built an C++ class called “TimeFcnt”, which has a method named “long sleep_cycle(long)”. The method takes the required period in
microsecond as argument and returns the actual task execution time in current cycle. The
typical usage of the class is shown in Figure 4-7.

```c
TimeFcnt clock; //Start clock; Use clock.reset() if clock exists
for(...) { //or other loop
    //periodic task here
    clock.sleep_cycle(1000000); //period of 1 second
}
```

**Figure 4-7 Example of Usage of TimeFcnt Class**

The functions were tested for 1000 cycles with period of 1 second, totally 1000
seconds. The average cumulative error on 20 runs is 0.008 second. This error is caused by
random error and it does not change significantly with the total number of cycles. For
comparison, using the method without error control (the second approach described in
this section), the same test yielded an error of 10 seconds, or 1.0%. The “TimeFcnt” class
has been widely used in the programs in this work.

### 4.5.2 Network Load Generator

The network load generator program, “loadgen”, was built to generate network
traffic to a specified host. The data is sent to the UDP Discard port (port number 9) of the
remote host. The usage of the program is shown as follow:

```
loadgen host Kbytes_per_second [time_in_second].
```

Among the command line arguments, “host” is the host name of the remote machine that
the data is sent to; “Kbytes_per_second” gives the data transmission rate; and the
optional argument “time_in_second” specifies for how long the traffic should be
generated. If “time_in_second” is omitted, the program will keep sending data until being interrupted by user.

The data is sent in bursts of 50 Kbytes with uniform time intervals between bursts. The time function described in the previous section was used in the program to accurately control the time intervals. For example, the following command:

```
loadgen america 1000 15
```

generates 1000 Kbytes/second traffic to the host “america” for 15 seconds. Since the data is sent in bursts of 50 Kbytes, the number of bursts in every second is 1000/50 = 20, and the time interval between bursts is 1/20 = 0.05 second.

### 4.5.3 Network Monitoring Display

The network monitoring display program was built using Java Swing GUI components, and it can be run on Windows or Unix machines that have a Java 2 platform. Figure 4-8 shows a screenshot of the display program.

When the display program is started, it opens an unused port and waits for the network monitoring program to connect. The host name and the port number are displayed on the title bar of the display panel. In Figure 4-8, the host on which the display is running is “ssx3” and the listening port number is 2000. The network monitoring program can connect to the display using the “-D” option:

```
```

It monitors the network traffic between host “agent” and host “america”, and sends bandwidth information to host “ssx3” at port number 2000 for display. After the connection to the display program is established, the monitoring program first informs
the display of the names of the two end hosts, which, as the result, appears on the upper left part of the display. Then the network monitor keeps sending the latest results of both utilized and available bandwidths to the display. These results are displayed on the mid-left part of the panel. The total bandwidth is simply the sum of utilized and available bandwidths.

Figure 4-8 Screenshot of the Network Monitoring Display
The percentage information and the historical curve are also shown on the display. The user can choose to display either the bandwidth used or the bandwidth available by selecting between the two radio buttons on the top-right part of the panel. In Figure 4-8, the “Show Used Bandwidth” option is selected, therefore the percentage usage of the bandwidth and the bandwidth usage history are displayed.

The percentage is shown both in digits and in a bar chart. The current percentage usage is 33% (4,100,448 out of 12,500,000). The curve on the bottom of the panel shows the history of the bandwidth usage. The whole curve shifts left every one-second with the most recent data is shown near the right border. The distance between two adjacent vertical grids represents 10 seconds.

The Java classes used to build the display are listed in Table 4-3. The display program can be invoked by issuing command “java NMD” in the directory where these Java classes reside.

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMD</td>
<td>The main class that starts the Network Monitoring Display.</td>
</tr>
<tr>
<td>NMDFrame</td>
<td>The main frame of the display.</td>
</tr>
<tr>
<td>HeaderCanvas</td>
<td>The part (upper-left) of display that shows the host names.</td>
</tr>
<tr>
<td>TextCanvas</td>
<td>The part (mid-left) of display that shows the bandwidth in text.</td>
</tr>
<tr>
<td>PercentCanvas</td>
<td>The part (mid-right) of display that shows the percentage.</td>
</tr>
<tr>
<td>PlotCanvas</td>
<td>The part (bottom) of display that shows the historical curve.</td>
</tr>
<tr>
<td>PlotData</td>
<td>The class that stores the data used by PlotCanvas class.</td>
</tr>
<tr>
<td>DataServer</td>
<td>The server class that receives and update bandwidth data.</td>
</tr>
</tbody>
</table>
4.5.4 Programs Using SNMP++ API Functions

Three accessory programs were built using the “SnmpppApi” functions. They provide some useful network information, and, in addition, they serve as testing programs for the SNMP++ API functions. The usage and description of the programs are listed in Table 4-4.

Table 4-4 Programs Using SNMP++ API Functions

<table>
<thead>
<tr>
<th>Program</th>
<th>Usage*</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostif</td>
<td>hostif host [snmp_options]</td>
<td>Get the names of all network interfaces of the given host.</td>
</tr>
<tr>
<td>hostin</td>
<td>hostif host -[io] [snmp_options]</td>
<td>Monitoring network traffic through each interface of the given host. The sum of incoming and outgoing data bytes is reported periodically, or they can be reported separately by using -i or -o option, respectively.</td>
</tr>
<tr>
<td>imon</td>
<td>imon [snmp_options]</td>
<td>Monitoring the network traffic through the specified network interfaces. A list of the host/interface pairs is defined and may be changed in the source code, imon.cc.</td>
</tr>
</tbody>
</table>

*[snmp_options] are the same as those in section 4.4.

The “hostif” program detects the network interfaces of a specified host. For example, the “hostif agent” command will give us the following results:

<lo0>
<hme0>
<hme1>

The “lo0” is the local loop-back interface. Host “agent” actually has two network interfaces: “hme0” and “hme1”. “hostif” can also be used to detect if the SNMP server is
properly running on the specified host. If the SNMP server is down, the program will
time-out and display an error message.

The “hostin” program goes one step further than “hostif”. It monitors the data
traffic through all the interfaces of the specified host. This can be useful if the user’s
interest is on one particular host. We have used this program to discover the physical
network connections between computer hosts and the switch. The switch in our lab has
24 ports and almost all of them are used, which makes it very hard to track down one
cable between a host and the switch. We used the “hostin” program to monitor the traffic
through each port on the switch, and, at the same time, generate traffic between different
hosts using the “loadgen” program. The interface can easily be found by matching the
traffic pattern from the results of “hostin”.

The “imon” program monitors the traffic on several network interfaces. The
interfaces may be on different hosts. The program is relatively simple because it does the
monitoring work without the knowledge of network topology. This program is mainly
used for tests that involve several machines.
Chapter 5  Experiments and Results

5.1 Chapter Outline

The network monitoring program we developed was tested over a local area network (LAN) environment. The experiments and the results are discussed in this chapter. The detail of the setup of the LAN test bed is given in next section. The setup will be frequently referred by the later sections.

Section 5.3 shows the tests of the SNMP polling results of a network interface. The experiment results are carefully examined with statistical analysis, including the accuracy of the network load generator and SNMP polling overhead.

Section 5.4 gives the experiment results for a communication path. The network topology discovery and path traversal are tested. The monitored bandwidth information is compared with the generated network load.

Different algorithms have been used for the bandwidth calculation of hub-connected networks and switch-connected networks. Two different sets of experiments have been done to examine the correctness of our implementation. The results are listed in Section 5.5 and Section 5.6, respectively.

5.2 Experiment Setups

The experiments were performed in the Laboratory for Intelligent Real-time Secure Systems (LIRTSS) at Ohio University. The network is a LAN system with several
computers connected by a 100-Mbps switch and a 10-Mbps hub. The machines have different platforms, including Unix (Solaris 7 and 8), Linux, and Windows NT.

As shown in Figure 5-1, one Linux machine (L1), two Solaris 7 machines (S1, S2), and four Solaris 8 machines (S3-S6) are connected to the switch. The speed of these network connections is 100 Mbps, or 12.5 Mbytes/second. Two other Windows NT machines (N1 and N2) are connected to the hub with a speed of 10 Mbps, or 1.25 Mbytes/second. The hub and the switch are connected through a 10 Mbps link.

![Figure 5-1 Experiment Setups of LAN Test Bed](image)

Most of the experiments were performed on the Linux machine L1. Both the network monitoring program and the network load generator were running on it. SNMP servers were available on L1, N1, N2, S1, S2, and the switch at the time of experiments. Such a network arrangement guarantees that for any single network connection, at least one of the two end machines has an SNMP server running, so that the network traffic on the link can be monitored. For example, even though there is no SNMP server on any one
of the machines S3-S6, the bandwidth between any of them and the switch can still be monitored by polling the SNMP server on the switch for the interface connected to that machine.

In the following experiments, the network monitoring program was started on L1 to monitor the network traffic of the system. The network load generator on L1 was used to created traffic from L1 to a designated host or hosts. The bandwidth information of a network interface, or a communication path between two hosts, or several different paths, is monitored and recorded. The results were analyzed against the pattern of generated traffic load to examine the correctness of our program.

### 5.3 Network Interface

To examine the correctness and accuracy of the SNMP polling results, experiments were done on a single network interface. Network traffic was generated using our network load generator, while the traffic through the network interface was monitored using the “imon” program described in section 4.5.4. The observed values were compared with the generated traffic.

In the experiments, the designated network interface is a 100 Mbps interface on one of the Solaris 7 machine. Five runs were performed at different times. For each run, different network traffic loads were used, starting with no generated load, followed by 100, 200, 300, 400, and 500, then 1000, 2000, 3000, 4000, and 5000 Kbytes/second loads. For each network load, we measured 2000 data points with 1 second interval between points. Therefore, for each network load, there are 10,000 data points. The data were averaged and the results are listed in Table 5.1.
Table 5-1 Experiment Results for Network Interface

<table>
<thead>
<tr>
<th>Generated Load (Kbytes)</th>
<th>Measured Traffic (Kbytes)</th>
<th>Traffic Less Background (Kbytes)</th>
<th>Traffic Percentage Error</th>
<th>Measured Number of Packets</th>
<th>Number of Packets Less Background</th>
<th>Average Packet Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9.55</td>
<td>0.00</td>
<td>22.83</td>
<td>0</td>
<td>22.83</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>111.65</td>
<td>102.10</td>
<td>2.10%</td>
<td>92.95</td>
<td>70.11</td>
<td>1491.08</td>
</tr>
<tr>
<td>200</td>
<td>213.91</td>
<td>204.36</td>
<td>2.18%</td>
<td>162.54</td>
<td>139.70</td>
<td>1497.93</td>
</tr>
<tr>
<td>300</td>
<td>316.91</td>
<td>307.36</td>
<td>2.45%</td>
<td>233.59</td>
<td>210.75</td>
<td>1493.36</td>
</tr>
<tr>
<td>400</td>
<td>419.52</td>
<td>409.97</td>
<td>2.49%</td>
<td>303.47</td>
<td>280.63</td>
<td>1495.95</td>
</tr>
<tr>
<td>500</td>
<td>521.34</td>
<td>511.79</td>
<td>2.36%</td>
<td>374.18</td>
<td>351.34</td>
<td>1491.61</td>
</tr>
<tr>
<td>1000</td>
<td>1032.62</td>
<td>1023.07</td>
<td>2.31%</td>
<td>723.28</td>
<td>700.45</td>
<td>1495.65</td>
</tr>
<tr>
<td>2000</td>
<td>2056.07</td>
<td>2046.52</td>
<td>2.33%</td>
<td>1423.16</td>
<td>1400.32</td>
<td>1496.54</td>
</tr>
<tr>
<td>3000</td>
<td>3079.48</td>
<td>3069.93</td>
<td>2.33%</td>
<td>2123.27</td>
<td>2100.44</td>
<td>1496.65</td>
</tr>
<tr>
<td>4000</td>
<td>4103.21</td>
<td>4093.66</td>
<td>2.34%</td>
<td>2824.98</td>
<td>2802.15</td>
<td>1495.96</td>
</tr>
<tr>
<td>5000</td>
<td>5129.27</td>
<td>5119.72</td>
<td>2.39%</td>
<td>3527.27</td>
<td>3504.44</td>
<td>1495.99</td>
</tr>
</tbody>
</table>

Both the number of bytes and the number of packets were measured. For example, with no generated network load, the average traffic measured is 9.55 Kbytes and 22.83 packets each second, while with 5000 Kbytes/second load, the traffic measured is 5129.27 Kbytes and 3527.27 packets per second.

The traffic measured at 0-generated-load was used as the background traffic. Such background traffic may come from the data communication of other programs, including the SNMP polling of our networking monitor and the responses of the SNMP daemon. The background was subtracted from the average values, which gives a more accurate measure of traffic due to the generated load. For example, the traffic of 5119.72 Kbytes and 3504.44 packets each second is caused by the 5000 Kbytes/second load we generated.
The measured traffic, including both the number of bytes and the number of packets, is plotted in Figure 5-2. Both plots show good linearity. The measured traffic (bytes) is slightly larger than the generated load, while the measured number of packets has a rate about 700 packets per 1000 generated load.

![Figure 5-2 Experiment Results for Network Interface](image)

The measured traffic (bytes) was compared with the generated network load, and the percentage errors were calculated and listed in Table 5.1. The error ranges from 2.1%
– 2.5%, with an average of 2.3%. The error is plotted in Figure 5-3 and it appears as a flat curve at the bottom. For comparison, the percentage error calculated with the measured values before removing the background was also shown as the curve at the top. The error is much larger when the generated load is small, and gets closer to the bottom curve when the load gets larger. It indicates that our choice of removing the background is reasonable.

![Figure 5-3 Percentage Error of Measured Traffic](image)

The measured traffic is about 2.1–2.5% larger than the generated load. A more careful examination of the traffic load generated by the load generator can reduce the error significantly. The size of generated load being used is the actual number of data bytes sent using write() system call. But it does not count the Ethernet, IP and UDP headers. Typically, an Ethernet header is 14 bytes long, an IP header is 20 bytes, and a UDP header is 8 bytes. For each packet, there are totally 42 bytes of extra data. The Maximum Transmission Unit (MTU) of our system is 1500 bytes. Therefore, for a packet
with MTU size of 1500 bytes, only $1500 - 42 = 1458$ bytes are real data. The overhead caused by the headers is approximately $42 / 1458 = 2.9\%$. Considering such overhead, the actual error of measured value is less than 1\%. Therefore, we may conclude that the network traffic reported by our monitoring program is correct and accurate.

Another interesting observation in these experiments is the average packet size (Table 5-1 and Figure 5-4), which is calculated using the number of bytes and number of packets measured. The average size ranges from 1491 to 1498 bytes, which is very close to the MTU size of 1500 bytes. The measured size is slightly smaller because there is a small portion of small-size packets.

![Figure 5-4 Average Packet Size](image)

Experiments were done to test the response of our network monitoring program when generated load approaches or exceeds the maximum capacity of the network link. The results showed different behaviors for switch and hub connected links.
Figure 5-5 shows the monitored network traffic of a 100 Mbps switch-connected link. The circle points are the observed data, and the solid line represents the generated load. Network load of 15000 Kbytes/second was generated from 100th second to 200th second. The maximum network capacity of the link is 12,207 Kbytes/second (shown in the figure as dashed line), therefore the link was overloaded during this period.

![Figure 5-5 Overloading Switch Connected Network Link](image)

The observed data points when the network was overloaded are unstable because the SNMP query and answer packets were delayed or lost when transmitted through the congested network. However most of the points lay around maximum bandwidth line and do give us the correct result. Some improvement can be done to reach more accurate results when the network is congested. One solution is to report that the network link is in
full capacity when there is no response for the SNMP query. An alternative way is to simply report to the resource manager that the host does not respond. There is a status field in the data structure for the communication between resource manager and network monitor, and this field can be used to report the situation. Another possible solution to eliminate the large fluctuation caused by the delay in SNMP polling is to average over the data of a certain number of most recent cycles. Figure 5-6 shows the result when the data of three cycles, current cycles and two previous cycles, are averaged. It gives more stable results comparing with the original data in Figure 5-5.

![Figure 5-6 Revised Results for Overloaded Network Link](image)

In contrast to a switch, a hub is much less reliable when it is congested. Figure 5-7 shows the measured network traffic for a 10 Mbps hub-connected link at different generated loads. It shows very good linearity until the generated load approaches the full capacity of the hub. The network link became congested when generated load reached...
1150 Kbytes/second. When more traffic was added, the congestion became worse. More SNMP queries or responses were lost, and the network monitor could not get an SNMP response for a long period of time, or even no response at all. A possible solution, which is similar to the one for the switch case, is to report that the network link is at full capacity or the host does not respond if the network monitor cannot obtain any SNMP response for certain number of periods.

![Graph showing the relationship between generated load and measured traffic](image)

**Figure 5-7 Maximizing Hub Connected Network Link**

In summary, our network monitor can accurately report the network traffic of a network link when the link is not overloaded. Extra steps need to be taken when the network becomes congested. For overloaded switch-connected links, with careful examination of the data, our network monitor may still correctly report the network
information. For 10 Mbps hub-connected network link, network bandwidth can be monitored up to 1150 Kbytes/second (92% of full capacity). Higher network load may cause delay or loss of the SNMP query/response, which makes the link unable to be monitored. However, such delay or loss can be used as an indicator of a congested network link.

5.4 Communication Path

A set of experiments was performed to observe the network traffic of the communication path between a Windows NT machine, N1, and a Solaris 7 machine, S1. The network topology information was loaded first when the network monitor started. Figure 5-8 shows the debugging information printed out by the monitoring program. It shows a list of all hosts, network devices and network connections, which maps the network topology shown in Figure 5-1. The information is in the format of “host <interface>”. For example, machine L1 is connected through network interface “eth0” to the switch at interface “FastEthernet0/2”.

The communication path between two hosts, S1 and N1, was successfully retrieved by our program using topology information. As shown in Figure 5-8, the path starts from host S1, interface “hme0”, goes through the switch (interfaces “FastEthernet0/3” and “FastEthernet0/14”) and the hub (interfaces “port0” and “port1”), and reaches the host N1 at interface “3Com EtherLink PCI”.

To check the correctness of the network bandwidth usage reported by our network monitoring program, network traffic was generated from L1 to N1 using the network load generator. Starting at 0 Kbytes/second for 120 seconds, we increased the amount of data sent by the load generator by 100 Kbytes/second each 60 seconds. After 360 seconds, the load generator was sending 500 Kbytes/second from L1 to N1. The entire load was eliminated at 420 seconds (Figure 5-9a).
The output of our network monitor exhibits a similar pattern to the amount of data being generated (Figure 5-9b). The reported value of used bandwidth is slightly larger than the generated traffic load due to the background traffic over the network and the overhead in the packet headers discussed in the previous section. The fluctuation and spikes seen in Figure 5-9b is caused by the fluctuation of data transmission over network and a slight delay in SNMP polling.
The experiment results show that our network monitoring program successfully find the communication path between the given host pair and correctly monitored the network traffic through the path.

5.5 Hub Connected Network

As discussed in the earlier chapters, a hub forwards data packets to all the connected hosts, not just the one for which a packet is destined. This affects the bandwidth of all hosts connected to a hub if data is sent to any host connected to the same hub. Our monitoring program considers this by summing the traffic through a hub when computing the amount of bandwidth used on any communication path through the hub. A test was designed particularly for the communication paths involving the hub to check how their network traffics affect each other.

The experiment was run to monitor the amount of bandwidth usage of two communication paths: S1 to N1, and S1 to N2. The two Windows NT machines, N1 and N2, are connected to the hub. Data was sent from the Linux machine to the two NT machines as shown in Figure 5-10(a-b). We started with no data being sent to either NT machines. After 20 seconds, we began to send 200 Kbytes/second from L1 to N1. 20 seconds later, we began to send 200 Kbytes/second from L1 to N2. After another 20 seconds, the traffic from L1 to N1 was reduced to 0. 20 seconds later the traffic from L1 to N2 was also eliminated.
The observed traffics for the two paths (Figure 5-10c-d) show a similar pattern, which are the sum of the two network loads generated, and the traffic to N2 from 40 to 80 seconds appears in the path between S1 and N1. The bandwidth usage doubles from 40 to 60 seconds when two generated loads overlaps. The results are just as we predicted.
5.6 Switch Connected Network

Unlike a hub, a switch only forwards packets to the host for which they are destined, not all the hosts connected to the switch. Therefore, our monitoring program treats switches differently than hubs. The traffic through a switch is not summed up. Instead, only traffic going to and from a particular host is considered when computing the amount of bandwidth being used.

An experiment was performed on two hosts connected by the switch. Traffic between S1 and S2, and between S1 and S3 was measured by our program. All three machines are Solaris machines and are connected to the switch. As shown in Figure 5-11(a-c), traffic of a similar pattern with the test of hub of were generated: 2000 Kbytes/second of traffic was generated at time 20-60 seconds from L1 to S2, while the same amount of traffic was sent to S3 from 40-80 seconds, and finally, traffic was sent to S1 from 100-120 seconds.

The observed bandwidth usages of two communication paths are shown in Figure 5-11(d-e). The load sent to S2 and S3 appears only in the path between S1 and S2, and between S1 and S3, respectively, while the load to S1 is present in both paths because S1 is in both paths. Data sent to S2 does not affect S3 and vice versa. The results shows that our network monitoring program has correctly implemented the data transmission mechanism of switches.
Figure 5-11 Experiment Results for Switch-connected Network
(a-c) Patterns of traffic loads generated by the load generator. (d-e) Measured traffic between hosts according to our network monitoring program.
Chapter 6  Conclusions and Future Work

6.1 Conclusions

This thesis has presented our approach to monitoring the network QoS in a dynamic real-time system. Network topology information is obtained from network hardware specifications, and real-time network performance of individual network host and device is gathered using SNMP polling. Real-time network metrics (communication bandwidth) are calculated based on the information. These accurate real-time network metrics are then sent to the resource manager to enable it to manage the QoS constraints of all programs.

Algorithms were developed to calculate the bandwidth usage information of a communication path between two hosts. Different algorithms are used for host pairs connected by different network devices (hub and switch). Details of the algorithms were discussed in this thesis. The algorithms were implemented using C++ and SNMP++, an SNMP tool. Several API functions and accessory programs were built during the development of the network monitoring software.

The programs were tested on a LAN environment. The results of experiments showed that the algorithms had been correctly implemented, and the results reported by the monitoring program were accurate.

In summary, a network monitoring mechanism was successfully designed and implemented to monitor the network QoS.
6.2 Future Work

This thesis has shown some preliminary work on network monitoring for the resource management project. Many features of the program can be improved for better performance. Possible future work includes:

- Measurement of network latency: together with the bandwidth information, this may give a more complete network metrics.

- Detection of network QoS violation: discovery of violations such as network congestions, lost of connections, can help the resource manager to detect and diagnose QoS violations caused by network.

- Dynamic network topology discovery: this can be useful for the real-time systems with dynamically changing networks, such as satellites, mobile networks.

- Distributed network monitoring: for a large network, distributed network monitoring can be more effective and more efficient.
References


