The University of Kansas

Information and Telecommunication Technology Center

Technical Report

Implementation of a Scalable Agent-Based Network Measurement Infrastructure to Improve the Performance of Distributed Application

Yulia Wijata
Douglas Niehaus

ITTC-FY99-TR-12161-03

Feburary 1999

Project Sponsor:
Information Technology Office
of the
Defense Advanced Research Projects Agency

Abstract

The rapid growth of computer networks has made the process of understanding the interaction among network components more challenging than ever. The increase in the size of the network is accompanied by more demanding use of the network by distributed applications that critically rely on the network to function well. Consequently, monitoring the health and stability of the network has become crucial. Numerous efforts have been devoted to measure the performance of the network for the purpose of network management or performance evaluation through creation of measurement tools or network probes. However, there is not yet a standardized "measurement infrastructure" which offers the systematic control and management of measurement efforts and performance data. The Internet is a classic example of an under-measured and under-instrumented network. This work addresses the issues related to creating a scalable and extensible network measurement infrastructure. These include an extension to NetSpec to support continuous network monitoring and the implementation of an agent-based monitoring system. In particular, the measurement infrastructure will be used to capture the network state to improve the performance of a distributed application.

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

Today's computer networks are experiencing massive growth in terms of quantity, quality and complexity of services offered. The components of the network are distributed by nature and typically comprise many different devices from different vendors, many operating systems, multiple distributed file systems and span many administrative domains. This distributed environment makes it harder than ever to manage the network and understand the interaction between components in the network.

Researchers have spent considerable effort to analyze and assess the performance of today's computer networks. In the early stages, analytical modeling and simulation are used to study the feasibility and evaluate the performance of various network architectures. However, simulation and analysis alone cannot capture every important aspect of a complex network. This calls for real-time testing and monitoring of operational networks. Doing so allows the capture of the state and many characteristics of the network for further study. Knowledge of the state of the network is important in several aspects:

- *Performance assessment* – Understand the current level of utilization of the network and service level expectation.
- *Problem solving* – Detection of bottlenecks, over/under-utilized resources.
- *Design and planning* – Understand the performance trends and traffic characteristics.

- *Network-aware Application* – Allow an application to adjust its use of network services in response to changes in network state.

Numerous efforts have been devoted to monitor and probe the network for the purpose of network management or performance evaluation. Many measurement techniques and tools are widely available and used. However, they are typically disjoint from each other and do not allow simultaneous and thorough investigation of the whole system. Attempting to do so usually results in an *ad hoc* solution which is not scalable and extensible. There is yet a need for an *integrated* framework for network testing, evaluation and monitoring which collects measurement data from various places and layers throughout the network.

NetSpec [29] aims to provide the generic solution for network testing in one integrated tool. Its control structure allows scalable, flexible and extensible measurements and testing to be carried out. However, the current framework lacks the capability of continuous monitoring and thus, limits the responsiveness of the tool to capture the network state for real-time execution and analysis. NetSpec's batch execution of test and measurement only produces report summaries at the end of the test. Introducing the ability to gather performance data *during* the testing will expand the types of data that we can collect and allow more detailed analysis by looking at the *interesting* events from various layers during the test.

This thesis will discuss the modification made to the NetSpec's control structure and daemon framework to allow continuous data collection. We will also identify and implement some examples of measurements and analysis which are now possible with the new framework. A measurement daemon based on the Simple Network Management Protocol (SNMP) can be used to continuously monitor network devices. Another measurement daemon collects data from the operating system kernel for detail analysis of the subsystems in the operating system.

Capturing network state, however, goes beyond carrying out network measurement and collecting the data. It requires specific knowledge about the network elements and topology and systematic placement and control of probes at *interesting* places throughout the network and end hosts. The raw data need to be filtered, correlated, organized for storage and finally processed to provide meaningful analysis. In order to do all this, we need a set of tools to automate the process of controlling network testing and measurement, collection and storage of performance data, and finally correlation and presentation of performance data.

To demonstrate the capture of network state, we will describe a monitoring system developed in the MAGIC-II [31] testbed. It consists of a set of monitoring agents which control NetSpec experiments and process the data collection. The information about the state of the network captured by the monitoring system is used to improve the performance of a distributed image storage system by allowing dynamic server selection where a client can dynamically select the server that can deliver the best performance.

The next chapter discusses related work in the areas of network testing and monitoring systems. Chapter 3 discusses the extensions made to NetSpec to allow continuous monitoring. Chapter 4 provides some examples of measurement daemons developed under the new framework. Chapter 5 describes the monitoring system to demonstrate the capture of network state. Finally, Chapter 6 presents our conclusion and discusses future work.

Chapter 2

Related Work

A considerable amount of work has been done in the area of network monitoring in general. Various types of monitoring tools and measurement methodologies have been developed in a quest to gain more information about the networks and eventually understand the complex behavior of the computer networks. Most of these monitoring efforts are targeted toward the assessment of performance or resource utilization and for detecting problem areas or misconfiguration in the network. More recent efforts show interest in providing the knowledge about the network to the distributed applications in order to achieve adaptation or dynamic reconfiguration.

No matter what the objectives are, the task of monitoring a large computer network is difficult, especially with the recent emergence of high speed wide area networks and distributed applications. This chapter will provide some background concepts and consider the state-of-the-art of network monitoring systems. Finally, we will also look at some other research efforts that specifically focus in capturing the state of the network and different areas of applications where it can be beneficial.

2.1 Network Measurement and Monitoring Tools

Les Cottrell of the Stanford Linear Acceleration Center (SLAC) provides an updated list of the monitoring tools [11]. Most of these tools are public domain and popular because they can be easily customized to fit specific monitoring requirement. However, there is not yet a standardized framework to control and manage the network monitoring itself. Section 2.1.1 describes some of the basic monitoring tools existing today. Section 2.1.2 presents a more integrated approach that leads to structured and well organized network monitoring.

2.1.1 Basic Tool Taxonomy

The National Laboratory for Applied Network Research (NLANR) [39], through the Cooperative Association for Internet Data Analysis (CAIDA) [4] provides a *taxonomy of measurement tools* [5]. Basically, the currently available tools can be categorized into several broad categories based on the measurement methodologies and goals. Here, we will examine some basic measurement strategies and identify the existing tools.

Ping

All *ping* based tools take advantage of the Internet Control Mechanism Protocol (ICMP) function, ICMP_ECHO, to check the availability of remote hosts and assess the latency in an IP based network. More sophisticated tools were derived from the original and simple ping, such as *Nikhef Ping* and *fping*.

Hop-by-hop Characterization

Van Jacobson utilizes the Time To Live (TTL) field in the IP packet to detect the path taken by a packet in *traceroute* [23] and *pathchar* [24]. A packet is directed to each router along a path without actually knowing the path, by setting the IP TTL field from 1 to n until the ultimate destination is reached. Upon receiving a packet with an expired (0) TTL, the hop generates an ICMP Time Exceeded response back to the source, thus identifying the hop and its round trip delay.

Packet Pair Analysis

The packet pair effect is described by Van Jacobson in [22]. Essentially, if two packets are caused to travel together such that they are queued as a pair at the bottleneck link, with no packets intervening between them, then the inter-packet spacing will be proportional to the processing time required for the bottleneck router to process the second packet of the pair, thus indicating the speed of the bottleneck link. This method has been used to measure the bottleneck bandwidth of a link without actually generating too much traffic (e.g. *bcprobe* [7]).

Packet Capture

Tools in this category capture packets as they go through a network interface and dump the headers of the packets for detail analysis that match some filtering condition. The *libpcap* [21] library provides some portable packet dumping routines which were originally used in *tcpdump*. Some other tools, such as OC3Mon [2], use specialized hardware to perform the packet capture more efficiently.

SNMP

The Simple Network Management Protocol (SNMP) [8] is a widely accepted protocol to perform network management functions. This lightweight protocol is mostly useful to collect status and statistic information from network devices, such as routers and switches. The emergence of the Remote Network Monitoring (RMON) [45] standard promotes the use of SNMP in traffic monitoring. Multi Router Traffic Grapher (*mrtg*) [40] is a popular tool based on SNMP which collects and graphs vital statistics from a router.

Load Generator

Tools such as *ttcp* [38], *netperf* [25] and *nettest* [44] provide the mechanism to perform throughput benchmark by generating some traffic pattern into the network and measuring the bulk transfer capacity. Note that while the idea of NetSpec originated from performing similar measurements, the latest development emphasizes creating a framework for performing distributed network measurement and is not limited to traffic generation and throughput measurement.

Traffic Flow Profiling

The underlying measurement mechanism in flow analysis is actually the same as in packet capture analysis. However, tools in this category focus more on characterizing information on a per flow basis. Cisco's *NetFlow* [10] collects flow statistics from Cisco's router. *NeTraMet* is a public domain tool to perform flow monitoring for accounting purpose.

2.1.2 Integrated Monitoring System

While numerous efforts have been focused in the development of individual tools to assess the performance of the computer network, only few have looked into creating a large scale network measurement infrastructure which controls the complex process of doing network measurements and organizing the performance metrics. This section will explain some recent efforts which are geared toward creating such a framework.

2.1.2.1 Objective Driven Monitoring

Mazumdar and Lazar [33] describe a framework for real-time monitoring of broadband networks using an objective-driven approach. The basic unit for the monitoring system is the a system of sensors, a knowledge base, and an inference engine. Objective driven monitoring allows the activation of sensors for data collection and abstraction based on a set of objectives. The objectives are derived from the quality of service requirements for real-time traffic control and operator submitted queries. The methodology of objective-driven monitoring for selective activation of sensors is implemented as a set of rules in the knowledge base of the monitor.

2.1.2.2 Network Analysis Infrastructure (NAI)

The Measurement and Operations Analysis Team of the National Laboratory for Applied Network Research (NLANR) is working on a Network Analysis Infrastructure (NAI) [34] which focuses on the integration of measurement activities at various sites in the vBNS (very high Backbone Network Service) network. The main goal is to define performance metrics and define analysis objectives in order to gain insights and knowledge about the inner workings of the Internet.

The infrastructure is currently considering these four areas of measurement data:

- **Passive workload profile assessments** are conducted by OC3Mon/Coral [2] monitors that collect packet header traces and perform flow analysis on the traces.
- **Active performance measurements** assess host reachability, packet losses, and throughput measurements. Performance parameters will measured from within the network, i.e. via probes deployed within the infrastructure itself.
- **SNMP/MIB based statistics data** are collected from participating routers.
- **Stabilities and status of Internet routing** are obtained by collecting the Autonomous System (AS) number from the Border Gateway Protocol (BGP) routing.

The overall analysis infrastructure requires sets of machines and tools with various layer of functionalities, including: data collection engine, local analysis/abstracting, data transfer to collation site, central analysis/abstracting, result cataloguing and aggregation and user interface for presentation of data.

2.1.2.3 National Internet Measurement Infrastructure (NIMI)

The NIMI Project [1] is an undergoing project that aims to facilitate the development of a large-scale *measurement infrastructure* for the Internet. One of the principle roles for such an infrastructure is to measure *vital signs* of the network, such as throughput, latency, and packet loss rates.

Its architecture is patterned after Paxson Network Probe Daemon (NPD) [42] which was used to perform the first large-scale measurements of end-to-end Internet behavior. An important aspect of the architecture is that it completely separates the tasks of *making measurements* , *requesting measurements* , *analyzing results* , and *configuring probes.* In keeping with this separation of tasks, NIMI architecture defines several entities:

- **NIMI daemons** (*nimid*) act as end-points for a set of measurement tools by providing communication and scheduling interface. Another important feature of the NIMI daemon is the secure authentication of the measurement delegation task.

- **Configuration Point of Contact (CPOC)** allows a site to configure a collection of NIMI daemons in a single place.

- **Measurement Point of Contact (MPOC)** allows a coordinated set of measurements to be configured at a single location.

Currently, the NIMI probe uses *traceroute* to perform hop-by-hop measurement, *TReno* to measure throughput and *Poip* to measure latency and packet loss of a given path. However, the plug-in design of the daemon allows any tool to be included as part of a measurement suite.

2.1.2.4 Windmill: Extensible Passive Measurement Tools

Unlike other measurement infrastructures that use active monitoring, Windmill [32] supports passive performance measurement of application-level protocols through the use of protocol reconstruction and abstraction-breaching protocol event monitoring.

To accommodate performance and extensibility, Windmill's software is split into three functional components:

- A dynamically generated protocol filter matches the underlying network traffic against a dynamically compiled filter.

- A set of abstract protocol modules provide both efficient implementations of target protocol layers and interfaces for accessing normally hidden protocol events.

- An extensible experiment engine provides a mechanism for the loading, modification and execution of probe experiments and provides the interfaces for the storage and dissemination of experimental results.

2.2 Other Efforts to Capture the State of the Network

Much of the current network measurement efforts are focused in collecting long term performance data to assess the performance trend and to identify bottlenecks in the network. The works described in this section use the knowledge obtained in network monitoring for different purposes. Section 2.2.1 describes some mechanism to optimize distributed systems by performing event-driven monitoring. Section 2.2.2 explains some recent works in the area of network-aware application where monitoring information is used by networked application to adapt to the dynamic condition in the network. Both categories of monitoring efforts are more oriented toward creating a *snapshot* of various events and conditions in the network and producing immediate analysis.

2.2.1 Tuning the Performance of Distributed Applications

NetLogger The NetLogger methodology [13] allows real-time diagnosis of performance problems in complex high-performance distributed system. The package includes the tools for generating precision event logs that can be used to provide detail end-to-end application and systems monitoring; a Java agent-based system for managing large amount of logging data; and tools for visualizing the log data and real-time state of the distributed systems. These tools are especially used for analyzing the performance of the distributed storage system in transferring large amount of data to a remote visualization client. The tools have been invaluable for diagnosing problems in networks and in distributed systems code.

2.2.2 Network Aware Applications

Globus The main objective of the Globus [18] toolkit is to build an Adaptive Wide Area Resource Environment (AWARE), an integrated and dynamically changing metacomputing environment. The unified resource information service provides a uniform mechanism for obtaining real-time information about metasystem structure and status. The rule based approach allows application to make late decision binding based on the current resource property.

Management of Application QoS Florissi and Yemini [16] proposes QuAL (Quality Assurance Language), a language for describing QoS constraints and monitoring. The specification is compiled into run time components that monitor the actual QoS delivered. Communication Monitoring Processes (CMPs) monitor QoS at network and application levels and maintain a QoS MIB.

Sumatra Sumatra [43] is an extension of Java language that supports resource aware mobile programs. It is used to built network-aware mobile programs that can use mobility as a tool to adapt to variation in network characteristics. Experiment with a distributed latency monitor tool shows that on-line network monitoring and adaptive placement of shared data structures significantly improve performance of distributed applications on the Internet.

2.3 Summary

This chapter has described some of the related work in network monitoring. Public domain monitoring tools exist in great abundance, however there is still no definite standard in this area, thus making collaboration and comparison of data difficult. Recent efforts in large scale testbeds such as VBNS or by organization such as NLANR show strong interest in creating a general network measurement infrastructure. This work described in this thesis particularly uses agent based technology and NetSpec to achieve a similar goal.

Chapter 3

Using NetSpec as a Monitoring Tool

NetSpec is a distributed network performance measurement tool designed and developed at the Information and Telecommunication Technology Center at the University of Kansas to provide convenient and sophisticated support for experiments evaluating the function and performance of wide area network. This chapter will briefly describe the design, architecture and components of NetSpec. Then it will discuss the modification made to the current framework to allow continuous monitoring and some implications caused by the extension.

3.1 NetSpec Design and Architecture

The following design criteria for NetSpec is specified [28]:

- **Scalability** – The framework must be able to support large scale experiment as networks generally carry hundreds of flow simultaneously.
- **Flexibility** – The framework must be flexible enough to handle both passive and active measurements.
- **Reproducibility** – The results of the experiments need to be reproducible, assuming the state of the network did not change.
- **Integration** – Measurements and tests need to be integrated in a seamless manner.

- **Extensible** – Addition of new components should be anticipated.

3.1.1 Motivation

NetSpec was developed mainly because conventional tools for conducting network performance experiments, such as `ttcp` [38] and `netperf` [25], lack the capability to perform large scale network level experiments involving a set of point-to-point connections. Moreover, the existing tools do not allow the seamless integration of measurement with traffic generations and thus, making extension of test types difficult.

3.1.2 NetSpec Components

NetSpec has undergone several revisions since the first idea was conceived. Here we will only describe the architecture for the latest and current version of NetSpec, namely version 3.0.

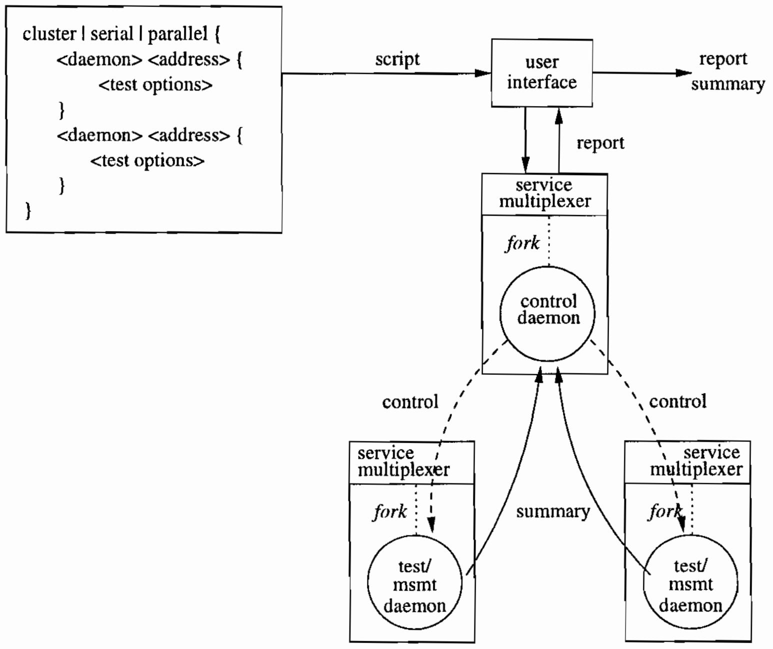


Figure 3.1: NetSpec High-Level Architecture

Figure 3.1 shows the components of the current NetSpec framework. Each of the components has to performs the following functions:

- User Interface
  - Read the script and pass it on to the control daemon.
  - Generate the protocol commands for the daemon invocation.
  - Accept the summaries of the experiments and present them to the user.
- Service Multiplexer

  Provide a single access point for invocation of all NetSpec daemons
- Control Daemon
  - Parse control protocol and parameters.
  - Control the invocation of other NetSpec daemons.
- Measurement/Test Daemon

  Perform specific measurement or testing functions.

All NetSpec experiments are described by the experiment script provided by the user. The script serves the following purposes:

- Identify the nodes involved in the experiment. A node is a generic term for a NetSpec daemon which can be a control, test, or measurement daemon.
- Identify the role assumed by each node.
- Define the test parameters for each measurement/test daemon.
- Describe the relationship among nodes and in so doing the topology of the experiment.

3.1.3 NetSpec Daemon Structure and Protocols

Probably the most challenging design aspect of NetSpec is creating a control framework which meets all the design goals described above. In order to achieve those goals, NetSpec provides a generic structure for a daemon implementation. Both the controller and measurement/test daemon are equivalent from a connection and protocol perspective. The generic structure allows the NetSpec framework to remain flexible and extensible. Adding a new type of test daemon requires no change in the control structure.

NetSpec uses a text based protocol to control the execution of the nodes involved in an experiment. The Remote Control and Information Protocol (RCIP) [27] imposes some sequence of phases that must be undergone by each daemon during execution. These phases mimic the phases of a typical network connection. In addition, RCIP also supports some administrative commands for control purposes. Table 3.1 summarizes the commands supported by RCIP and a brief description of actions taken when the command is invoked.

Command	Action
setup	Allocate resources.
open	Establish connection.
run	Start data transfer or measurement.
finish	Finish data transfer or measurement.
close	Close connection.
tear down	Free up resources.
report	Send report summary.
reset	Reset to initial state.
kill	Stop execution of daemon.
parameters	Accept parameters.
config	Return configuration information.

Table 3.1: RCIP commands and Execution Phases of a Daemon

A NetSpec daemon generally has two types of parsers :

1. A RCIP parser which parses protocol commands and invokes actions associated with the command, and

2. A parameter parser which parses test parameters to control test behavior.

The control daemon is an exception to this rule. Since the controller does not have any test parameters associated with it, it does not need the parameter parser. Moreover, the control daemon has a custom RCIP handler which is responsible for distributing control across multiple nodes according to the Distributed Control Language (DCL) specification [26]. Sections 3.1.3.1 and 3.1.3.2 will describe the implementation details of these two parsers.

Program 3.1 shows the pseudo-code for the generic NetSpec daemon structure.

Program 3.1 Generic NetSpec Daemon's Skeleton

```
/* parses option */
...
/* set up signals */
...
/* setup control connections */
...
while (no_error) {
   rcip_handler();
}
/* clean up */
```

3.1.3.1 RCIP handler

The RCIP handler is implemented as a state-based lexer which receives commands from the controller and invokes the appropriate action. A daemon's execution phase is essentially defined by the state of the lexer as it receives command invocation from the controller. Figure 3.2 summarizes the structure of the RCIP handler.

Notice that the daemon-specific parameter parser is invoked at the initial phase *before* it proceeds with the phases specified in Table 3.1. The parameter parser is responsible for building the data structures which contain the options describing the test/measurement being carried out.

3.1.3.2 DCL parser

As a special case of NetSpec daemon, the control daemon has some well-defined actions in response to RCIP command invocation because it is responsible for ensuring the synchronous operations of the daemons under its control. The core of the control daemon is comprised of the DCL parser which accepts the typical RCIP commands as defined in Table 3.1.

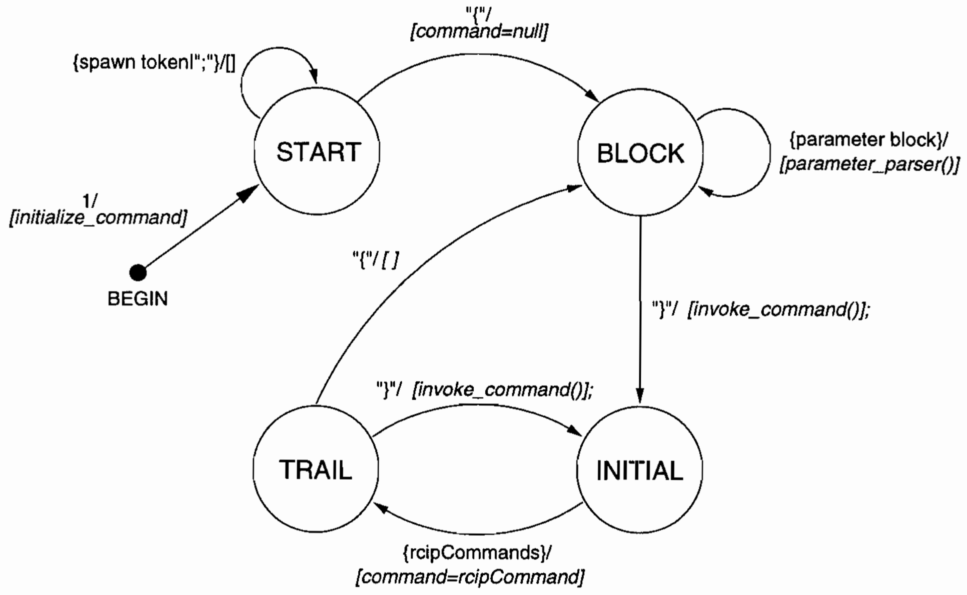


Figure 3.2: RCIP Handler Lexer State Machine

The DCL parser collects the list of the controlled test or measurement daemons and keeps them in a linked list. As it goes along, it establishes control channel to each daemon and passes the appropriate section of the experiment script. After the script is completely parsed, the controller begins to distribute the RCIP commands to each peer. Figure 3.3 shows the sequence of RCIP commands which the control daemon distributes to its peer.

The controller accepts three types of execution construct, namely: serial, parallel and cluster. Each execution type governs the behavior of the controller during its *run* phase. The order of RCIP commands sent to peer daemons dictates the phases of the peers and thus the behavior of the daemon. Figure 3.4 illustrates the behavior of the controller for each type of execution construct during the controller's *run* phase.

3.2 Extending NetSpec to Allow Continuous Monitoring

The flexibility and extensibility of NetSpec makes it a powerful tool to do system/network testing of any kinds in an integrated manner. However, the batch-execution style adopted by NetSpec only permits collection and analysis of results at *the end* of the

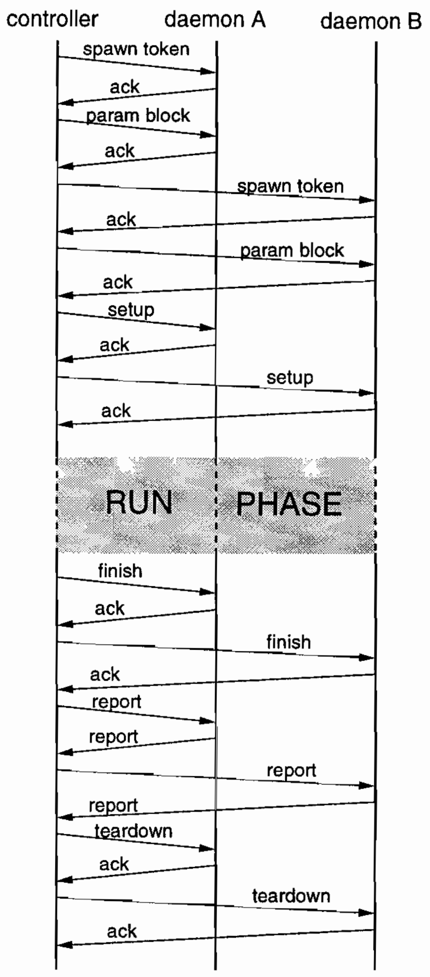
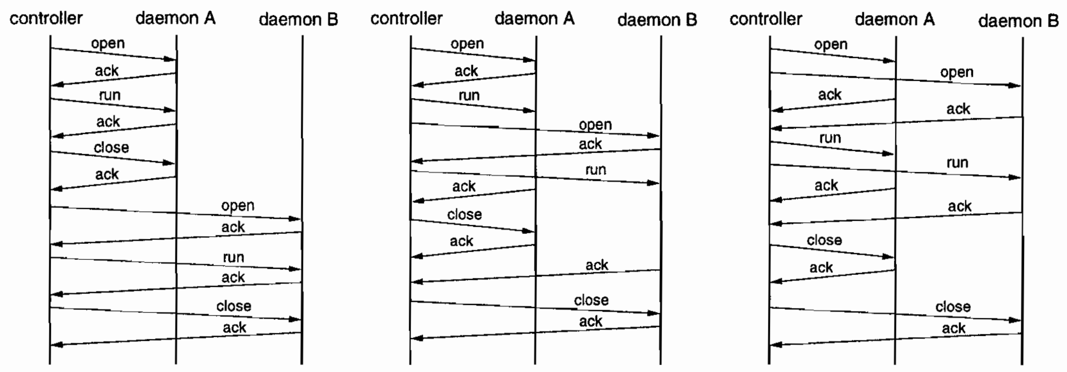


Figure 3.3: Distribution of Control Protocol

test. This limitation prohibits NetSpec to be used in a testing environment where more real-time execution and on-the-fly analysis are needed. Problems can also arise for the types of measurement which produces a large amount of data during the *run phase* of the experiment.

In the current framework, it is difficult to capture the dynamic characteristics of the system because only the statistical *summary* of the testing result is presented. Moreover, it is also difficult to correlate the data collected at various layers of the system without having a standard way of collecting and interpreting the events from different layers of the system.

To address this issue, an extension to NetSpec which allow continuous stream of



(a) serial (b) parallel (c) cluster

Figure 3.4: Different Types of Execution Constructs

data to flow from the test/measurement daemon *during* the course of an experiment is needed. With this extension, we are opening a window of opportunities to use NetSpec in a more real-time environment and for collecting data with a finer granularity.

The following sections will describe the modification and extension made to the current NetSpec framework to allow such a capability.

3.2.1 Design Criteria

The following design criteria have been set as a guideline in designing the modification to NetSpec. In addition, the design should always conform to NetSpec's design philosophy listed in Section 3.1 above.

- Integrated with NetSpec

  The framework should fit seamlessly with the rest of NetSpec's structure.

- Optional

  This continuous data generation should be an *optional* feature of a NetSpec daemon and should not impose any change to the current NetSpec daemon implementation.

- Generic

  The method should be applicable to a larger class of applications outside Net-

Spec.

- Unobtrusive

  Generation of events or intermediate reports should not interfere with the testing at hand.

- Standard format

  The framework should produce data in a standard format, hence, allowing fair comparison and interpretation of flows from multiple data streams.

3.2.2 Modified Architecture

To incorporate the continuous report generation at the daemon level, we will minimally need two additional entities in the NetSpec framework:

Report Collector Server (Daemon) which collects data from NetSpec test/measurement daemons.

Report Channel which connects a NetSpec daemon to the report collector server.

Note that the report collector server needs not be within the NetSpec framework. Any server which conforms to the protocol used by the NetSpec daemon should be able to read data from the report channel. For conciseness, the report collector server will also be referred to as a report daemon. On the other hand, the report channel is closely integrated with the NetSpec framework. It must be set up by the NetSpec control structure and becomes an integral part of a NetSpec daemon.

There are two options to handle the reporting capability of the daemon depending upon the entity who controls the report channel's setup.

Option 1: The reporting feature is part of the DCL protocol understood by the controller and hence, the report channel setup is invoked by the controller.

Option 2: The reporting feature is a standard NetSpec daemon parameter. In this case, the controller is not aware of the existence of such a report channel.

Note that in both options, the report channel is integrated with the daemon framework and is managed by each daemon. The first option requires modification of the DCL and RCIP protocols while the second option requires almost no change at all to the NetSpec protocol. However, making the reporting feature a daemon's parameter implies that each daemon must be able to parse the option and hence, reduces the generality of the NetSpec daemon's framework. With careful design and investigation of the current framework, the modification of DCL and RCIP protocols can be implemented with minimal effort while maintaining the generic nature of the daemon's framework.

Figure 3.5 shows the modified architecture. If we compare this figure with the old NetSpec architecture shown in Figure 3.1, we will notice the two additional entities mentioned above. NetSpec daemon can send report Protocol Data Unit (PDU) via the report channel to the report daemon. The report daemon accepts these PDUs and presents them to the user space in a human readable format.

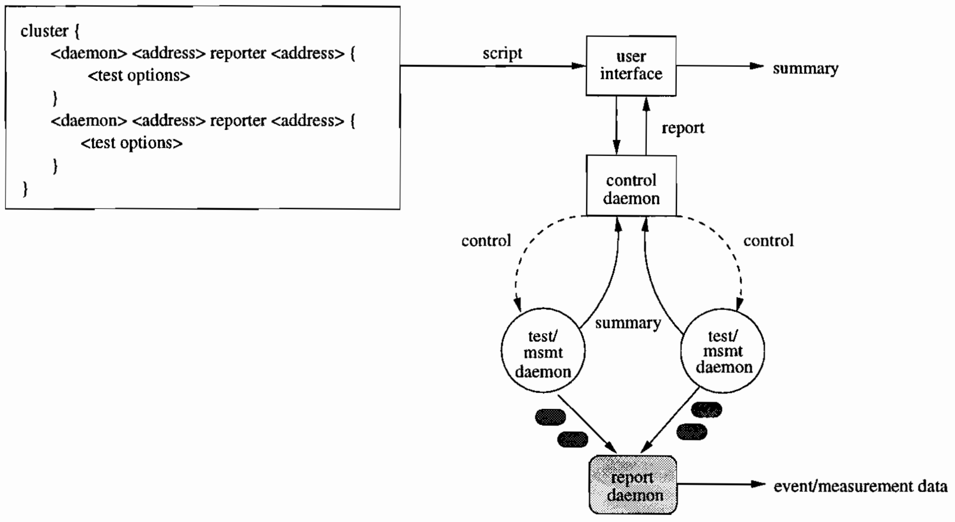


Figure 3.5: Modified Architecture with Inclusion of Report Channel

Figure 3.5 also shows the modification made to NetSpec script's format. Basically the script now includes the address of the reporter daemon with which a particular NetSpec daemons can interact. Since the continuous data generation is an optional feature of a NetSpec daemon, omission of the reporter address implies that the daemon will *not* generate any intermediate report during the course of the test or measurement.

Program 3.2 outlines the modified syntax for the block structure. NetSpec's block structure separates the parts which need to be understood by the controller and parts which are meaningful only to the test/measurement daemon:

```
[controller's semantics] {
    [daemon's semantics]
}
```

Since it is already argued above that the controller must be responsible for invoking the creation of the report channel, the reporting option should be placed outside the daemon parameter block. The following sections will describe the modifications made to the DCL and RCIP protocols and the process involved in the report channel creation and the report daemon structure.

Program 3.2 Modified block syntax

```
<daemon> <daemon_address> [reporter <reporter_address>] {
    <parameter block>
}
```

3.2.3 Modification to DCL Protocol

As mentioned in section 3.1.3.2 above, the controller invokes the creation of NetSpec daemons and establishes the control channels to its peers. To incorporate the reporting capability discussed in the modified architecture above, the DCL parser in the controller must parse the information about the reporter and pass the address of the reporter to the daemon so that a report channel can be created by the daemon to the specified reporter. Program 3.3 shows the modification made to DCL lexical analyzer and parser.

The report channel needs to be established as early as possible in the lifetime of a daemon so that report PDUs can be generated in the early phases of daemon execution. Although most of the time, report PDUs will be generated during the run phase of the daemon execution, the design should not limit itself to only that particular phase.

Program 3.3 Modified DCL lexical analyzer

```
dcl.l:
   "reporter"  return REPORTER;

dcl.y:
   peer
     : IDENTIFIER peerIpAddress reporterIpAddress
     {
      /* send spawn token */
      ...
      /* send replicate spawn token */
      ...
      /* send reporter token */
      writeLine("reporter", &currentPeer);
      /* send reporter address */
      writeLine($3, &currentPeer);
     } ...

   reporterIpAddress
     : REPORTER IPADDRESS ':' INTEGER
       {
      *$$ = '\0';
      strncat ($$, ipAddr2NumStr(str2IpAddr($2)), 255);
      strncat ($$, ":", 255-strlen($$));
      strncat ($$, $4, 255-strlen($$));
       } ...
```

To achieve this effect, the report channel needs to be established in the initial phase of the daemon execution. By looking at Figure 3.3, the ideal place to set up the report channel is before the controller sends the parameters block to the its peer daemon. This is done by sending the keyword `"reporter"` followed by the address of reporter in the form of `HOST:PORT`. Figure 3.6 shows the addition of these messages to the protocol. Note that if the reporter option is omitted, no additional protocol message will be sent to the peer daemon, and thus the controller behaves like a normal NetSpec execution without the reporting feature.

What will happen at the other end of the control channel as a response to the new protocol messages will be described in the following section.

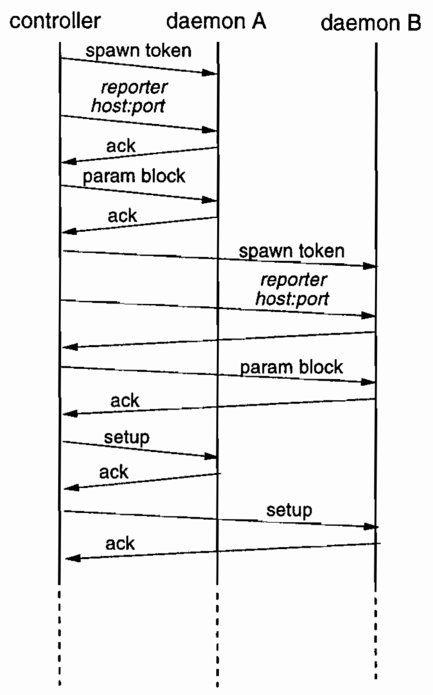


Figure 3.6: Modification to the DCL Protocol

3.2.4 Modification to the Daemon Framework

3.2.4.1 RCIP Handler

A daemon knows that it needs to establish a connection to a reporter when it receives the keyword `"reporter"` during the initial phase. This requires a slight change to the original RCIP parser shown in Figure 3.2. Basically, the handler must capture the keyword and set up the socket to the reporter. This can be achieved by adding a new state in the lexer in which the reporter address is collected and the report channel is setup. Figure 3.7 shows the modification.

3.2.4.2 A Case for UDP

The report channel can be implemented as either a TCP or UDP socket. While TCP is reliable in delivering the messages, the end-to-end control and the 3-way handshake during connection establishment and termination adds a considerable amount of overhead and thus can be obtrusive to the test being conducted. Since the reporting facility only requires a very lightweight protocol, UDP is much more suitable in this case be-

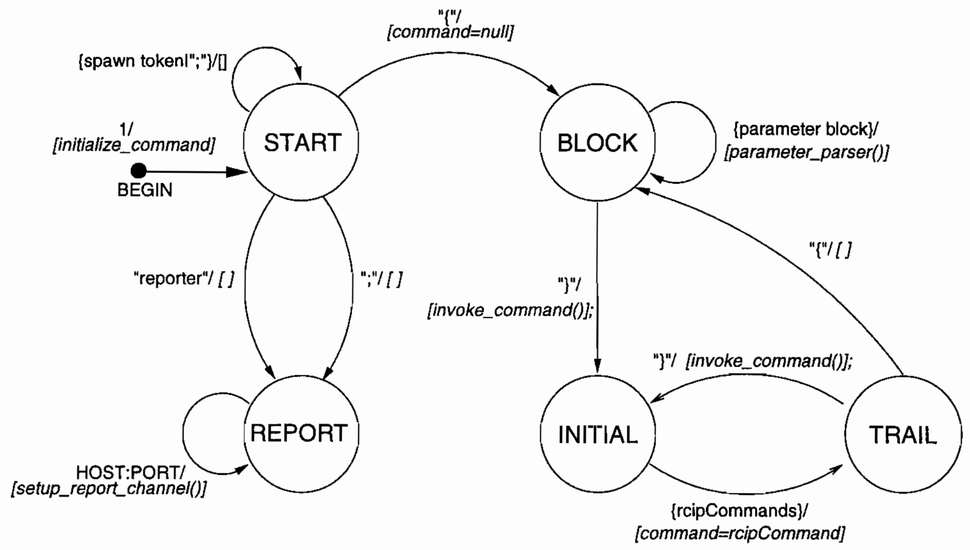


Figure 3.7: Modification to the RCIP Handler State Machine

cause the connectionless datagram service offered by UDP provides a really fast way to deliver message without incurring too much overhead.

3.2.4.3 Report PDU

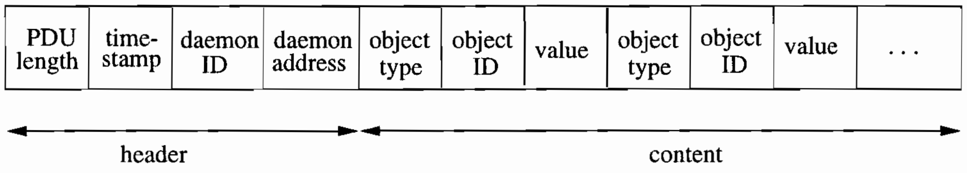


Figure 3.8: Report PDU format

Every intermediate report generated by the daemon is encapsulated in the report PDU as shown in Figure 3.8. Each PDU consists of the header fields and content fields. The fields in the header of the PDU can be explained as follows:

PDU length: the length of the PDU in bytes.

Timestamp: the timestamp of the creation of the PDU in

```
struct timeval {
     int sec;
```

```
                int usec;
            }
```

format, where `sec` and `usec` are the number of seconds and milliseconds since epoch, respectively.

Daemon ID: the name of the daemon which generates this PDU.

Daemon Address: the address of the daemon in form of `host:port`

The content of the PDU consists of one or more objects which are defined as a data unit generated by a NetSpec daemon. An object has a name and value of a supported type. An object is described in the `object_t` data structure shown in Program 3.4 where:

Object type is the type of the object. There are five basic types supported by the framework:

- An `integer` is a 4-bytes signed integer in the range $[-2^{31}, 2^{31} - 1]$
- An `unsigned integer` is a 4-bytes unsigned integer in the range $[0, 2^{32}]$.
- A `boolean` has the same representation as an integer but can only have a value of 0 or 1.
- A `double` is a 8-bytes double-precision floating point.
- A `string` is a null-terminated array of characters.

Object ID is a string representing the name of the measured object, and

Value is the value for this particular object corresponding to the specified type.

3.2.4.4 Report API

To ensure standard format of reporting facility, we need to provide a reporting Application Programming Interface (API) for the daemon. In our case, the API can be as simple as providing a function which handles the creation of report PDU and fills it

Program 3.4 Object data structures

```
#define INT_T 0
#define U_INT_T 1
#define BOOLEAN_T 3
#define DOUBLE_T 5
#define STRING_T 6

typedef union {
     int intval;
     unsigned int uintval;
     double doubleval;
     char *strval;
} objectVal_t;

typedef struct {
     int type;
     char objId[MAXOBJIDLEN];
     int len;
     objectVal_t val;
} object_t;
```

with the user's data and writes it to the report channel. The structure of the report PDU will be described in the following section.

Program 3.5 shows the pseudo-code of function `nsLogWrite()`. The logging function can accept an arbitrary number of objects to be packed in one report PDU. The reason behind this scheme is to reduce the number of report PDUs sent by allowing more than one object to share the same timestamp. The function creates an empty PDU and fills it with the header fields which consist of a timestamp and daemon identifier. Then it encodes or serializes the objects and writes it to the socket.

A NetSpec daemon must first create the objects before passing them to the `nsLogWrite()` function. We provide a function to create an object of each supported types. Program 3.6 shows the example functions to create an object of type `integer` and `string`. Each of these function create an object and fill in the name, value and type.

Since the data representation can vary from one computer architecture to the next, the eXternal Data Representation (XDR) [35] has been used for transporting the report

Program 3.5 Report API

```
void nsLogWrite (int numObjects, ...) {
   /* allocate space for PDU */

   /* create timestamp */

   /* fill PDU header with timestamp, own id */

   /* serialize each object into PDU */

   /* write the header of the PDU to socket */

   /* write the content of PDU to socket */

}
```

PDU to the reporter. XDR provides a mechanism to describe and encodes data for transfer between machines in a heterogeneous environment. It has been used in Remote Procedure Call (RPC) and Network File System (NFS) to ensure portability and data sharing across multiple machines. Our framework only uses some standard data types of all the data formats supported by XDR. These data types have been described in Section 3.2.4.3.

3.2.5 Standalone Reporter Structure

In order to meet the design criteria which mandates that the reporting facility should be generic enough, the new framework does not strictly dictate where the reporter server resides. Applications other than NetSpec can create a custom reporter server which could do processing specific to the applications' needs. For example, a reporter server can simply do post-processing of the reports into a format which can be used by other tools such as NetLogger [13] or NetAlyze [41]. This section describes the general structure and requirements of a standalone reporter which desires to collect NetSpec continuous data stream. To avoid confusion, the terms used in this section are as seen from the point of view of the reporter server. The terms *server* and *daemon* are

Program 3.6 Report API

```
ptr2Object_t createIntObject (char *id, int val) {

    ptr2Object_t ptr2Object = (ptr2Object_t) malloc(sizeof(object_t));
    ptr2Object->type = INT_T;
    memset(ptr2Object->objId, 0, MAXOBJIDLEN);
    strncpy(ptr2Object->objId, id, MAXOBJIDLEN);
    ptr2Object->len = sizeof(int);
    ptr2Object->val.intval = val;

    return ptr2Object;
}

ptr2Object_t createStringObject (char *id, char *str) {

    ptr2Object_t ptr2Object = (ptr2Object_t) malloc(sizeof(object_t));
    ptr2Object->type = STRING_T;
    memset(ptr2Object->objId, 0, MAXOBJIDLEN);
    strncpy(ptr2Object->objId, id, MAXOBJIDLEN);
    ptr2Object->len = strlen(str);
    ptr2Object->val.strval = str;

    return ptr2Object;

}
```

used interchangably when referring to the reporter, while NetSpec's test/measurement daemons which generate the performance data are referred to as *clients.*

The basic functions of the reporter server are fairly simple. The reporter binds itself to a certain address and waits for connection from one or more clients. Once a connection is establish, the reporter server accepts report PDUs, decodes the header, deserialize the objects, performs the necessary post-processing, and produces a suitable format of the report.

Since the NetSpec daemon uses XDR format to transfer data, the reporter must decode or deserialize the content of the PDU by following the XDR mechanism as well. When a report PDU arrives, the reporter first decodes the header and determine the number of objects in the PDU. Then for each object, it determines the type and

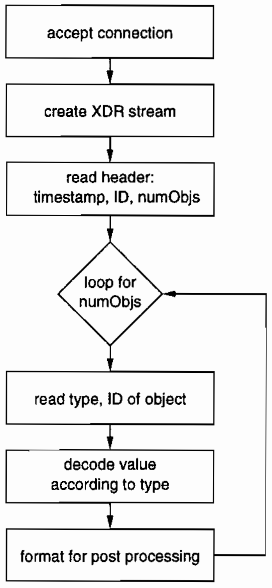


Figure 3.9: Reporter Algorithm in Decoding Report PDU

decodes the value accordingly. This algorithm is shown in Figure 3.9

3.3 Summary

This chapter has briefly described the design philosophy of NetSpec, a distributed network performance evaluation tool. It then describes the extension made to NetSpec to allow continuous or long term monitoring which permit test or measurement daemon to generate data during the course of the test itself.

Chapter 4

Monitoring Daemons Implementation

Several kinds of NetSpec test or measurement daemons have been developed recently for different purposes. Among the notable ones are:

- Test Daemon [28] generates various kind of traffic and measures the bulk transfer rate at the transmitting and receiving end.
- ATM Call Generator Daemon [37] is used to evaluate the performance of ATM switches.
- CORBA Daemon [19] is used in performance benchmarking of CORBA objects.
- SNMP Daemon [47] queries SNMP agents to collect SNMP variables from a MIB.
- DSKI Daemon [46] collects data from the operating system through the Data Stream Kernel Interface [3].

The continuous monitoring feature discussed in this thesis can be useful in support of some types of the daemons described above since it allows daemon to collect data in a more detailed manner and be able to communicate this information with other entity in the system in real time. Particularly, we will describe the implementation and examples scenarios for the SNMP and DSKI daemons.

4.1 SNMP Daemon

The Simple Network Management Protocol (SNMP) [8] has been widely accepted as the standard protocol in network management. Most network devices and end hosts comply with the SNMP standard and provide network managers a standard way to monitor many aspects of a network. Because of this, we were motivated to implement a NetSpec measurement daemon that is capable of communicating with network elements by using SNMP. Monitoring a set of objects in a network device can then be described conveniently through a NetSpec script.

Before going further with the implementation details, Section 4.1.1 will briefly describe the fundamentals concepts in SNMP. Then, Section 4.1.2 will describe the architecture and implementation of the NetSpec SNMP daemon. Section 4.1.3 will define the parameters of the daemon and the format of the NetSpec script.

4.1.1 Simple Network Management Protocol

As specified in RFC 1157, the SNMP architecture consists of a collection of network management stations and network elements. Network management stations execute *management applications* which monitor and control network elements. Network elements are devices such as hosts, gateways, terminal servers, and the like, which have *management agents* responsible for performing the network management functions requested by the network management stations. The Simple Network Management Protocol (SNMP) is used to communicate management information between the network management stations and the agents in the network elements. To achieve its goal of being simple, SNMP includes a limited set of management commands and responses. The network management stations issue `GetRequest`, `GetNextRequest` and `SetRequest` messages to retrieve single or multiple object variables or to establish the value of a single variable*. Figure 4.1, taken from IEEE Network Management Architecture [36], illustrates this concept.

Each SNMP agent maintains a database of network management information, called

*A variable is a unit of managed information in the agent.

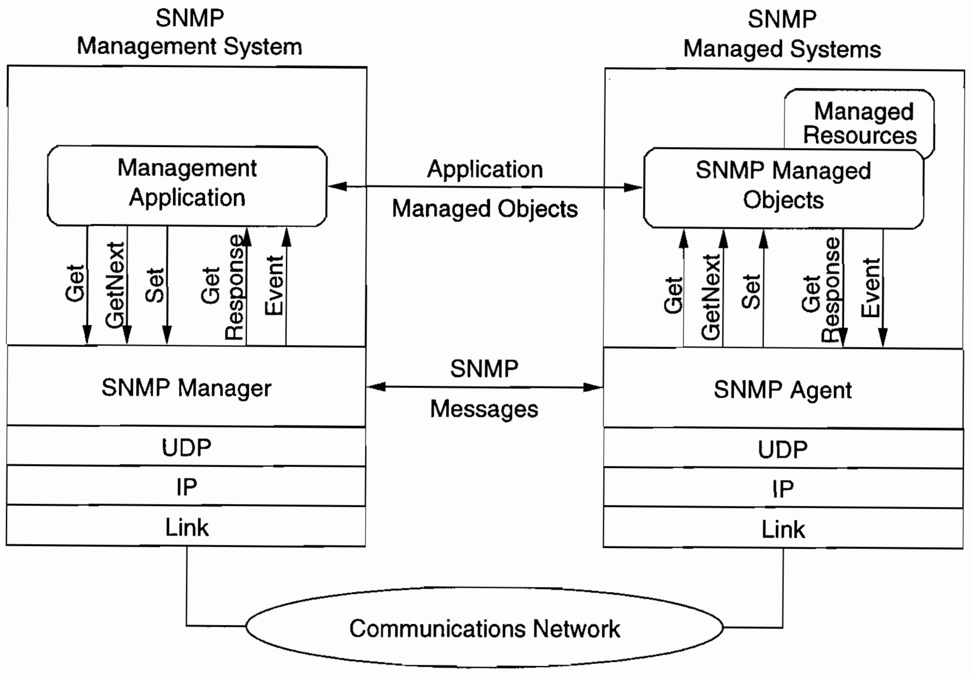


Figure 4.1: SNMP Architecture

the Management Information Base (MIB). The logical organization of the MIB is called the structure of management information (SMI), which is organized in a tree structure beginning at the root, with branches that organize the managed objects by logical categories. The MIB represents the managed objects as leaves on the branches. Figure 4.2 shows an example of the SMI for the system group. Each node in the tree is assigned a numeric value. A managed object is identified by its object identifier (OID) that is obtained by listing the numeric values of the traversed nodes in a string, separated by periods. For example, the variable `sysDescr` will have an OID of `1.3.6.1.2.1.1.1`.

4.1.2 SNMP Daemon Architecture

The function of the NetSpec SNMP daemon is to act as an SNMP manager that collects a set of managed information from an SNMP agent of a network device. The SNMP daemon establishes a communication channel with a SNMP agent through which it can send and receive SNMP PDU. Many of the management and monitoring operations involve a periodic collection of SNMP variables, which is why the continuous monitoring feature proves to be very beneficial for the operation of this daemon. We

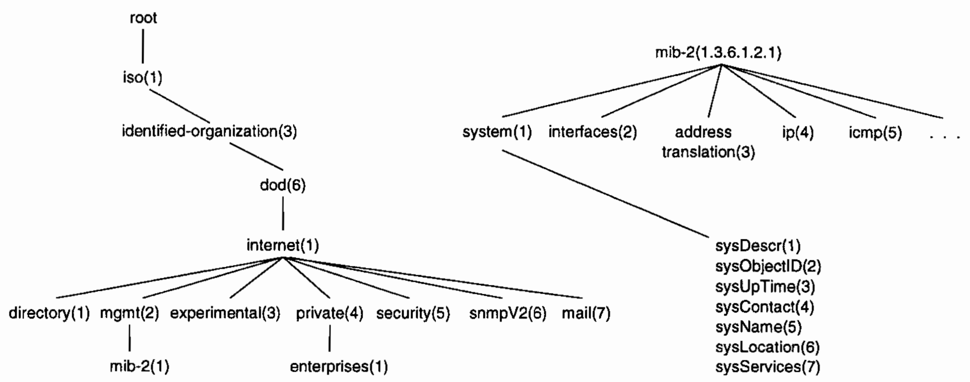


Figure 4.2: The System group

can set up the daemon so that it periodically polls the SNMP agent of a specific network device and generates the collected information on the fly. A report collector server can then use the information to monitor the latest state of that network device.

Figure 4.3 shows the architecture of the SNMP daemon and other entities it communicates with. The NetSpec script defines the network device and the managed objects to monitor. The SNMP daemon then sends the SNMP PDU(s) to the SNMP agent for that particular network device and waits for the response. If a reporter channel is specified for this daemon, it directly sends the data to the reporter for further processing. Otherwise, the daemon will store the data in a local buffer until it enters the report phase in which it will send the data to the user.

The SNMP daemon uses CMU-SNMP-v1 library from Carnegie Melon University. The library provides a basic API to create and interpret SNMP PDU as well as for creating a communication channel with an external SNMP agent. The library also requires an SMI which includes the descriptions of the managed objects. The SMI provides the mapping between a variable name and its OID. For objects that are not included in the SMI, the full path OID needs to be specified in the script.

The SNMP daemon implements the following NetSpec's daemon phases:

- Setup phase

    1. Initialize SMI
    2. Allocate buffers if necessary

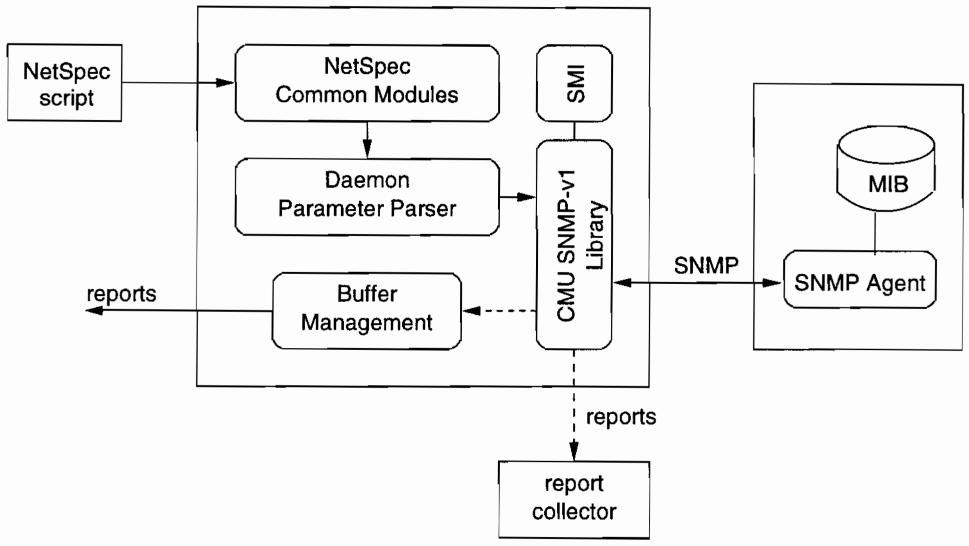


Figure 4.3: NetSpec SNMP Daemon Architecture

- Open phase
  1. Set up and establish SNMP connection with remote SNMP agent
- Run phase
  1. Send and receive SNMP PDU(s)
  2. If report channel exists, send data to report collector; else save in local buffer
- Close phase
  1. Close SNMP connection
- Finish phase
  1. Free SNMP PDU(s)
- Report phase
  1. If report channel exists, send summary; else, send the content of buffer to user
- Teardown phase
  1. Free buffer

4.1.3 SNMP Daemon Parameters

Table 4.1 describes the parameters for the SNMP daemon.

Parameter	Description	Value Syntax
device	Network device to be monitored	IPADDRESS
mibfile	The absolute path to the SMI	STRING
type	The type of collection	**oneshot** **periodic(duration=**INT**,period=**INT **)**
version	SNMP version to use	**v1 (community=** STRING **)**
operation	SNMP operation to perform and variable names	**get([var=**STRING**]*)**

Table 4.1: SNMP daemon parameters

For example, we want to poll the number of packets transmitted and received at the TCP layers every 10 seconds for 5 minutes and send the data to a report collector. We can describe that experiment in the following script:

```
snmp plato reporter plato:43813 {
     device = mauchly.ittc.ukans.edu;
     mibfile = "/usr/local/lib/mib.txt";
     type = periodic (duration = 300, period = 10);
     version = v1 (community = public);
     operation = get (var="tcp.tcpInSegs",
                      var="tcp.tcpOutSegs");
}
```

4.2 Data Stream Daemon

The function of the NetSpec Data Stream daemon is to collect data from the operating system kernel through the Data Stream Kernel Interface (DSKI). Since the nature of the information collected from the kernel are usually continuous, the provision for continuous monitoring in NetSpec is very relevant for the DSKI daemon.

Section 4.2.1 will provide a brief overview of the Data Stream Kernel Interface. Section 4.2.2 will describe the architecture and design of the NetSpec DSKI daemon and finally, Section 4.2.3 will describe the parameters and script format of the daemon.

4.2.1 Data Stream Kernel Interface (DSKI)

The main goal of the DSKI is to provide a standard and effective way of collecting data from the operating system kernel. The collections of data flowing from the operating system are generally referred to as *data streams.* The DSKI defines three types of data streams:

Active Data Stream allows us to record and later read a sequence of data items called *events.* This data stream provides a logging function which is called from selected points within the code of the kernel to log the occurrence of events of interest.

Passive Data Stream does not accumulate events. It stores a specific set of state information associated with an *object* within the system which the kernel updates with changes in the kernel state.

Cumulative Data Stream provides a way to gather statistics data related to events happening in the kernel in term of a *counter.* This type of data stream is useful for long term data collection where the detail of event traces is not required.

Figure 4.4 shows the architecture of the DSKI. In the kernel, related events, objects and counters are grouped in a *family* that is registered with the kernel. When an event occurs inside the kernel code, a timestamped event is created and added to the event queue of each user. A function that updates the counter can also be added to the kernel code for a cumulative data stream. The interface to the user is provided through the standard pseudo device driver interface. User can specify the type of data stream and read from the pseudo device.

4.2.2 DSKI Daemon Architecture

The DSKI daemon allows the collection of event traces and counter values from an instrumented operating system kernel. It provides a mechanism to describe the set of events and counters that a user wants to gather, as well as the duration and frequency of data collection. Figure 4.5 shows the structure of the DSKI daemon. As with the SNMP daemon, the presence of report channel causes data to flow to the report collector.

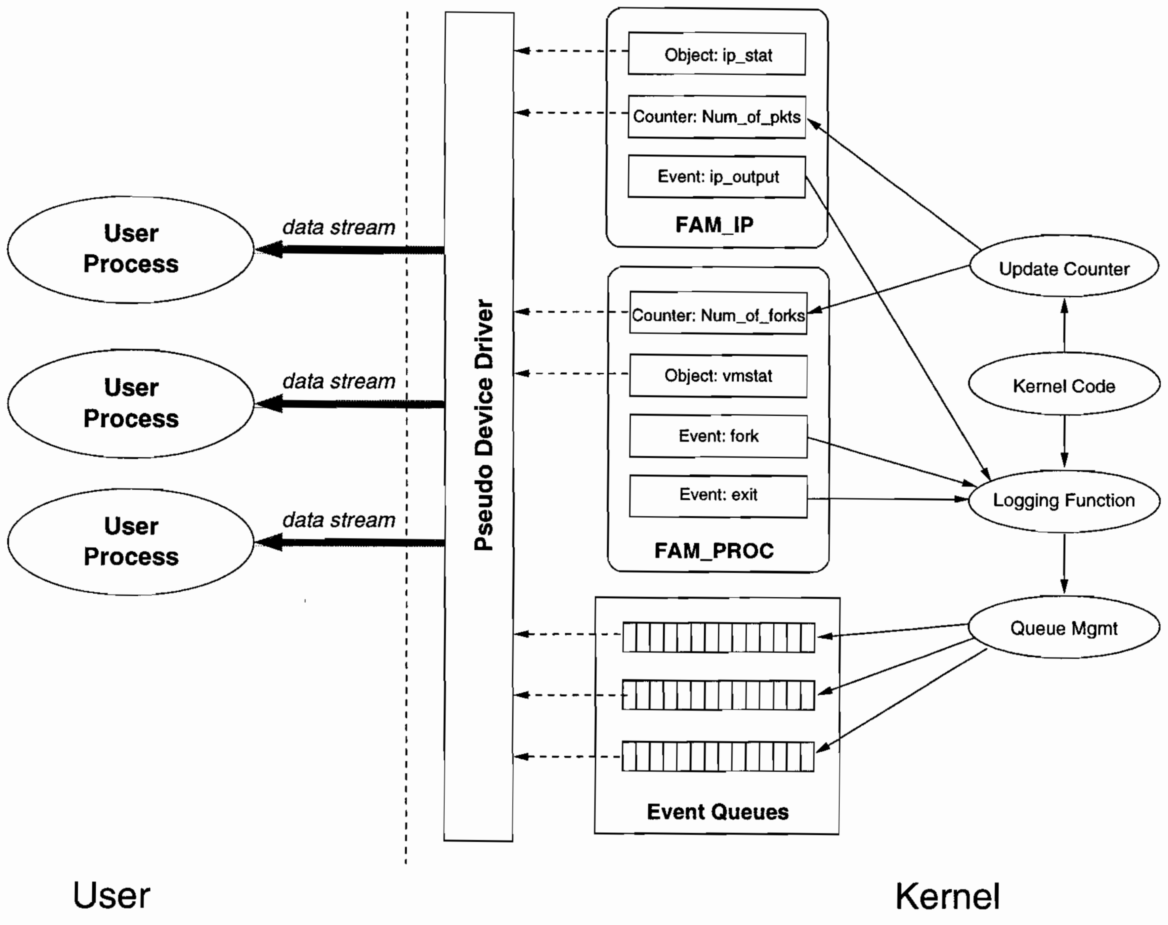


Figure 4.4: DSKI Architecture

The execution phases of the daemon are quite simple. It implements three of the NetSpec daemon's phases:

- Setup phase
  1. Open the pseudo device `/dev/dstream`
  2. Specify the type of data stream (ACTIVE or PASSIVE)
  3. Get event and counter names in each family for future mapping
  4. Select desired events
- Run phase
  1. Set collection duration or number of events to be logged

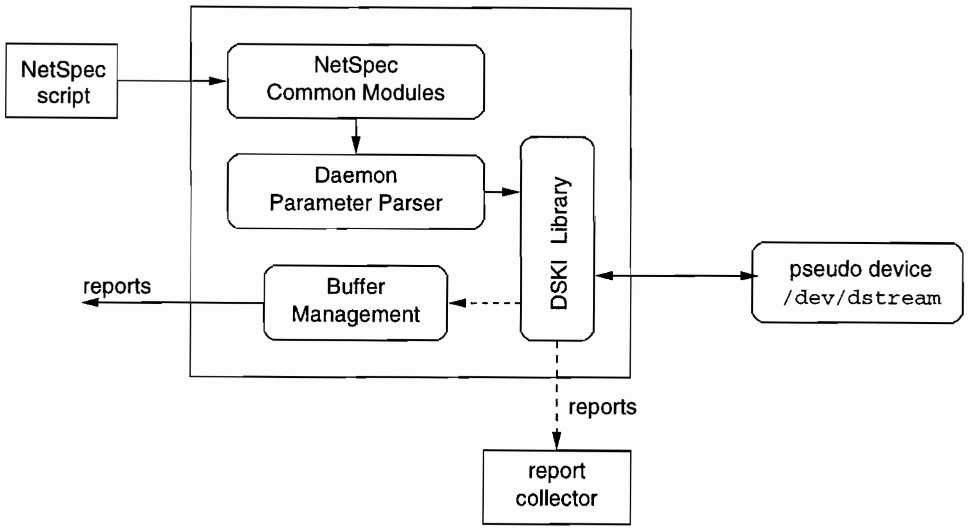


Figure 4.5: NetSpec DSKI Daemon Architecture

2. Start event logging in the kernel
3. Read event trace
4. If report channel exists, send data to report collector; else store in local buffer
5. Close pseudo device

- Report phase

1. If report channel exists, print summary; else print event traces

4.2.3 Data Stream Daemon Parameters

There are two types of scripts for the DSKI daemon, one for the active data stream and another for the cumulative data stream. A DSKI daemon can only gather data of one type of data stream. If more than one type of data stream flowing from an operating system kernel is desired, another DSKI daemon must be started on that machine.

The format of the script to collect event traces is shown in Program 4.1. The **numevents** parameter specifies the maximum number of events to collect. The optional parameter **duration** indicates the length of experiment in seconds. If a duration is not specified, the experiment will continue until the maximum count is reached. The user can filter out events from the trace by specifying the **param** value, which is a qualifier unique to a data stream family. For example, in the `ds_tcpip` family, **param** refers to the destination port of a packet. Only events generated by packets whose destination port matches the specified value of **param** will be included in the data stream. If the **param** option is not specified, the event trace comes unfiltered. For each *family*, a subset of events must be specified. If all events are to be collected, the keyword **all** can be used instead. Finally, *eventnames* is a coma-separated list of valid event names in a family.

The example shows an experiment to collect five event types from a `ds_tcpip` family at a machine called *plato* for 15 seconds or until 500 events are collected. In this family, `param` indicate the source port of a TCP packet.

Program 4.1 Script format to collect event trace and example

Script syntax:

```
dstream
      type = active (numevents = x, duration = y, param = z);
      family1 = set ( eventnames ) | all;
      family2 = set ( eventnames ) | all;
```

Example:

```
dstream plato {
      type = active (numevents=500, duration=15, param=42015);
      ds_tcpip = set (writeSocket, tcpOutStart, tcpOutEnd,
                      ipOutStart, ipOutEnd);
}
```

4.3 Characteristic Experiments

This section will provide an example for each NetSpec measurement daemon described in this chapter to demonstrate the benefits of having the continuous monitoring feature in NetSpec.

4.3.1 Monitoring ATM Switch Cell Rate with SNMP Daemon

This section demonstrates the application of the SNMP daemon to perform cell level measurements in an ATM network. Particularly, we would like to monitor the cell counts on the equivalent Virtual Circuits (VC) at the ATM switches belonging to the same connection.

SNMP is a convenient mechanism to perform such measurement because most networking devices support SNMP and maintain a MIB for network management purpose. Performance data are collected by querying the SNMP agent at each ATM switch.

Figure 4.6 shows the experiment set up in the MAGIC-II network. NetSpec test daemon generates a full blast traffic from host A, *armstrong.ukans.magic.net* to host B, *blackstone.tioc.magic.net*. A virtual circuit connection has been established between the two machines through switches X, *spork.ukans.magic.net* and Y, *hertz1.tioc.magic.net*, which are both FORE switches. Two NetSpec SNMP daemons poll the SNMP agent at the ATM switches for the cell count on a specific virtual circuit channel by specifying its port, VPI and VCI numbers. Cell counts at every second are sent to the report collectors for further analysis.

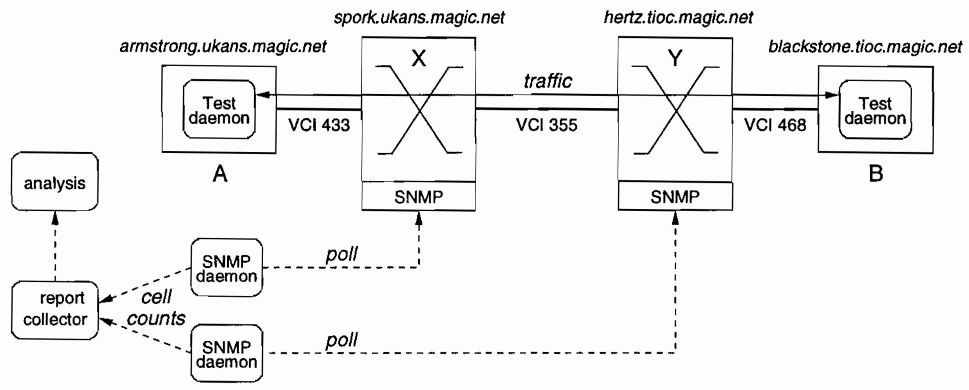


Figure 4.6: SNMP Experiment Set-up: Monitoring Cell Rates at ATM Switches

Figure 4.7 shows the cell rates at switches X and Y, respectively. As expected, we see a strong correlation between the rates at the two switches since they support the same connection. Both of the plots follow the same general pattern although not identical which may be caused by different cell delay variation at the two switches.

This experiment can be extended to include measurements at various levels such as at the network interface or TCP level by querying SNMP variables at different network devices. It demonstrates the strength of NetSpec in providing integrated control over different types of traffic generation and measurement. The following section will give

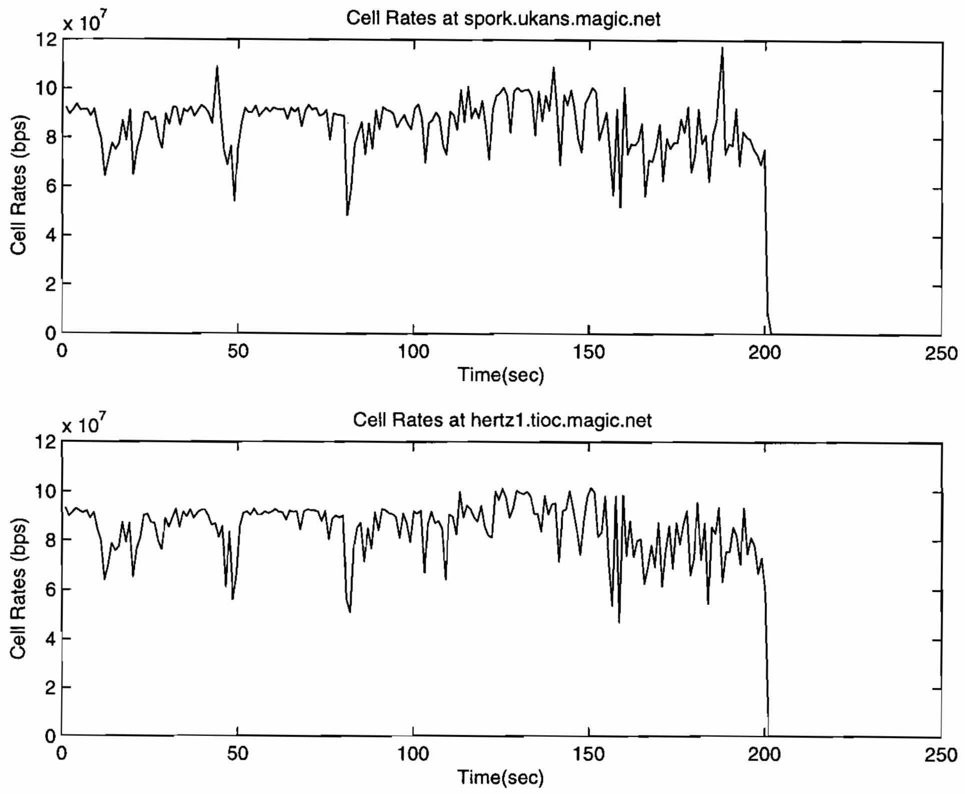


Figure 4.7: Cell Rates on the VCs at Two Switches Belonging to the same connection.

another example of measurement done at the operating system level which, when put under NetSpec control, can provide a rich set of performance data in addition to a more traditional performance data collection method.

4.3.2 Studying the TCP Congestion Mechanism with DSKI Daemon

Collecting detail event traces from an operating system kernel usually results in a large amount of data. By having the continuous data generation feature in NetSpec, we are saving system resources by immediately processing the event trace without having to store them in the system memory.

The experiment described in this section involves a performance evaluation of the TCP congestion mechanism, especially the evolution of the congestion window during adverse conditions. The traditional approach for this kind of experiment requires several independent processes to generate the traffic, collect and filter packet traces and analyze each packet's header. The NetSpec Test and DSKI daemons provide an integrated mechanism to describe and control the experiment.

Two DSKI families to collect event traces from an instrumented TCP/IP kernel module have been defined: a family for events related to *sending out* a packet and another for *receiving* a packet. Logging points have been inserted at crucial points in the stack as shown in Figure 4.8.

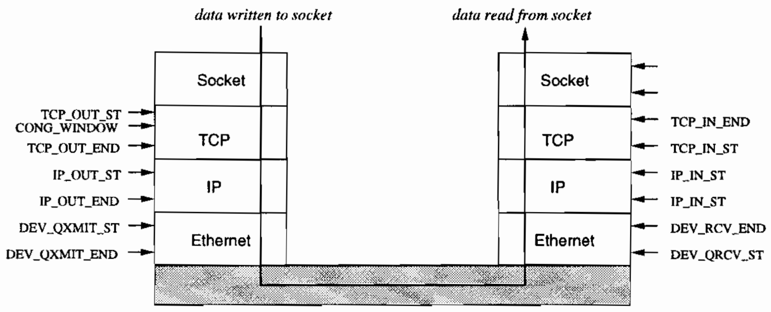


Figure 4.8: Instrumented TCP/IP Stack

We can obtain various kinds of detailed timing analysis by collecting event traces from an instrumented kernel such as this. For our purposes, we are interested in collecting events particularly at the TCP layer during transmission of a packet. Before the TCP layer passes a packet to the IP layer to be sent out to the network, the TCP congestion mechanism determines the effective congestion window size used. In our study here, we collect three events: EV_TCPOUT_ST, EV_CONG_WINDOW, and EV_TCPOUT_END.

The experiment set up is shown in Figure 4.9. The sender and receiver belong to the same ethernet segment in a LAN environment. The sender's kernel has DSKI configured and its protocol stack instrumented. The NetSpec Test daemon generates full blast traffic from the sender to the receiver. The NetSpec DSKI daemon collects event traces at the sender and sends the trace to the report collector located at another machine. In order to study the congestion mechanism, we must reinforce packet retransmission at the sender. The device driver for the physical interface at the sender has been modified to *drop one in every 500 packets* belonging to the test flow. Since the dropped packets never reached the receiver, the sender will not receive the ACK packets and this causes

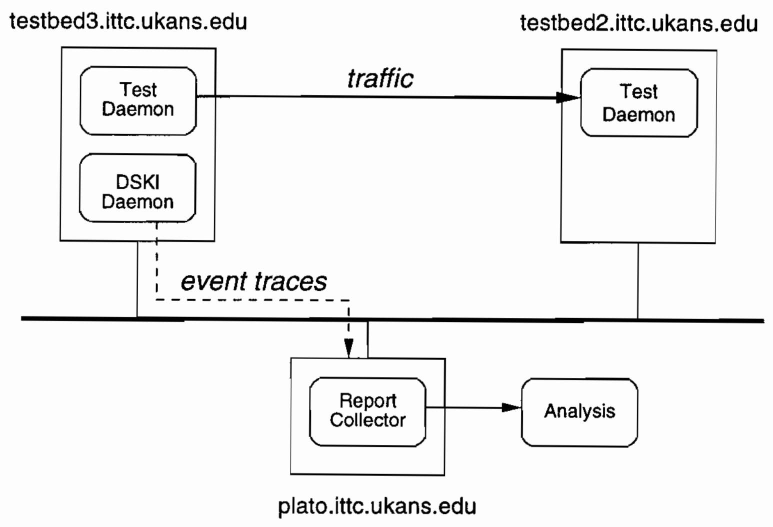
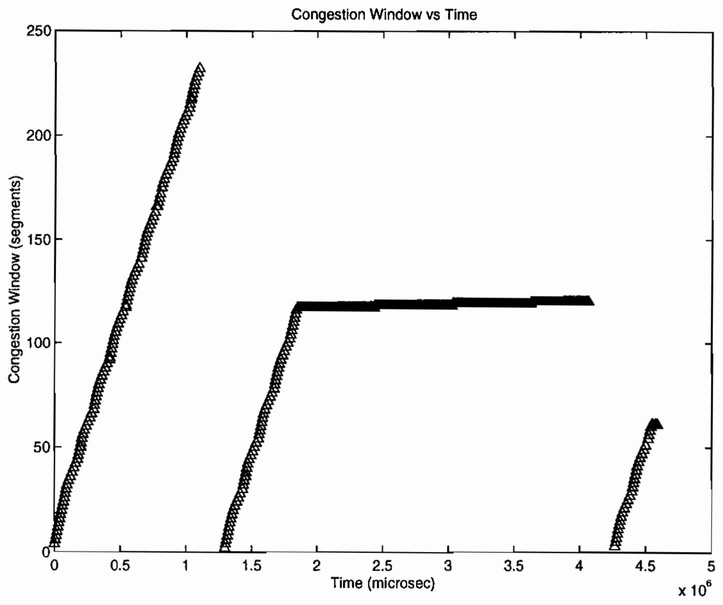


Figure 4.9: DSKI Experiment Set-up

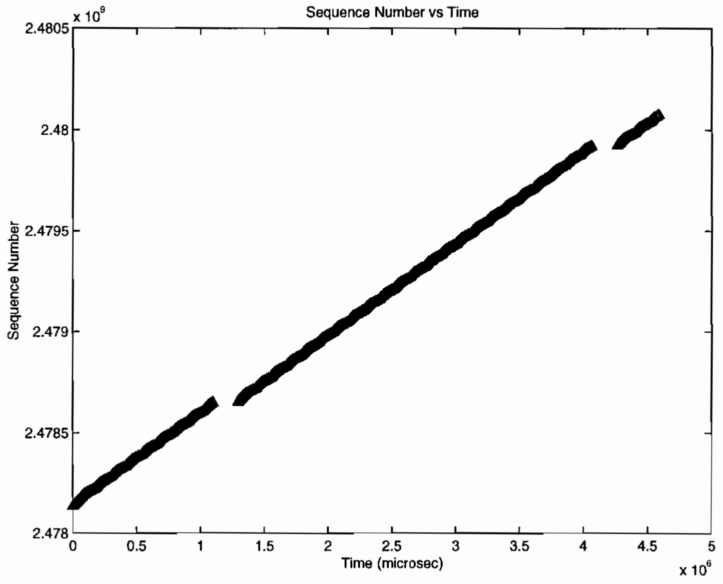
retransmission.

Figure 4.10(a) shows the evolution of the congestion window with time and Figure 4.10(b) shows the sequence numbers of the packets generated by TCP as time increases. These figures show the congestion avoidance mechanism used in TCP. At the beginning of the connection, the congestion window is increased linearly until it reaches the advertised receiver window. Retransmission is interpreted as an indication of congestion in the network. TCP attempts to avoid further congestion by slowing down its rate by reducing the congestion window size by half (this strategy is better known as *multiplicative decrease*). The *slow start* mechanism in TCP carries this idea one step further by reducing the congestion window to the size of MTU (1460 bytes for ethernet), and slowly increases it linearly until it reaches the threshold that has been set by multiplicative decrease.

In the next experiment, the rate of packet dropping at the device driver is increased to emulate a more adverse congestion situation. The network interface drops five consecutive packets for every 500 packets in the test flow. Figure 4.11(a) and Figure 4.11(b) show the evolution of congestion window and sequence number with time. Figure 4.11(a) particularly shows how slow start mechanism perpetuates the low transmission rate if loss keeps occurring in the connection. The congestion window is reset back to 1 MTU whenever retransmission occurs.

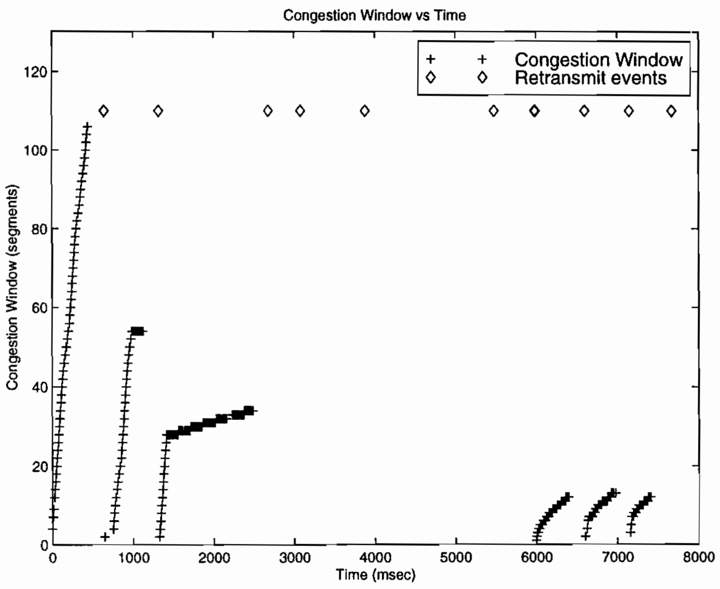


(a) Congestion Window

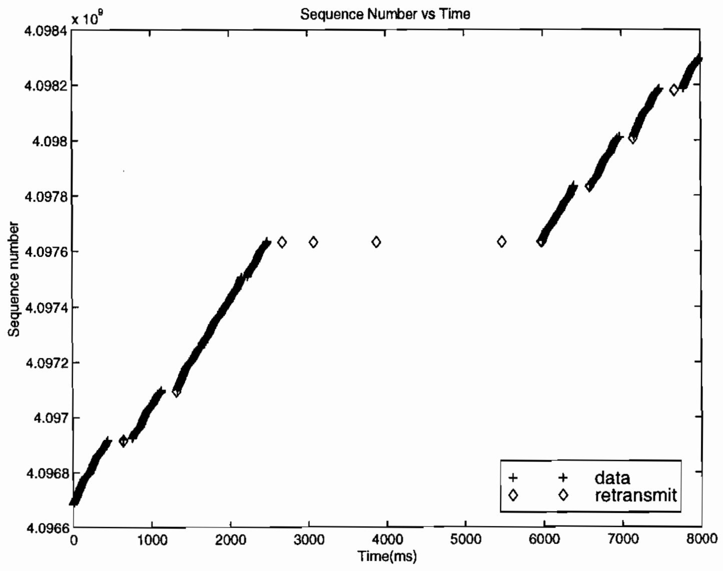


(b) Sequence Number

Figure 4.10: Evolution of Congestion Window and Sequence Number. 1 every 500 packets is dropped by the network interface.



(a) Congestion Window



(b) Sequence Number

Figure 4.11: Evolution of Congestion Window and Sequence Number. 5 consecutive packets out of 500 packets are dropped by the network interface.

4.4 Summary

This chapter has described the implementation of two types of measurement daemons which benefit from the new continuous monitoring feature of NetSpec: the SNMP daemon and the DSKI daemon. Two characteristic experiments have been presented for each type of daemon to demonstrate the usefulness of the continuous monitoring feature.

Chapter 5

Network Monitoring Efforts in MAGIC-II

5.1 Motivation and Background

The Internet is growing at such an overwhelming rate that understanding the interaction among the many Internet components has become more challenging than ever. Monitoring the network becomes a crucial task, especially with the emergence of distributed applications that critically rely on the network to function and which contribute to pushing the network to its limit while demanding a good quality of service.

Network monitoring provides important information to allow performance tuning of network operation by identifying *hot spots* and measuring the perceived quality of service from the user perspective. It can also be used to troubleshoot faults in the network by continuously keeping track of the status of the components in the network. Long term network monitoring has proven to be useful in understanding the trends in performance for future planning. Monitoring can also identify security problems or reinforce security mechanisms in the network. Recently, network monitoring has been taken to another level and challenge by using the knowledge about the network state to allow application level adaptation, more popularly known as network-aware applications.

Numerous efforts have been devoted to measure the performance of the network for the purpose of network management or performance evaluation through creation of measurement tools or network probes. However, there is not yet a standard "measurement infrastructure" which offers the systematic control and management of measurement efforts and performance data.

We approach this problem by treating the network as a distributed system and applying one of the common solutions to monitor a distributed system. We define a collection of software agents to provide integrated control of monitoring elements and collected information. Essentially, an agent is an autonomous entity that operates around a knowledge base (KB). In the context of this thesis, the responsibility of an agent is to *automate* one or more of the following tasks:

- Continuous monitoring of the components of distributed applications and relevant network characteristics
- Creation and control of network testing and measurement
- Collection and storage of performance data
- Correlation and presentation of performance data to application and/or user

In particular, the agent technology allows one to "wrap" legacy tools – in this case, network monitoring and measurement tools – with software and enable them to communicate via a common agent protocol such as the Knowledge and Query Manipulation Language (KQML) [15]. With this approach, the complex task of monitoring a large, distributed system can be decomposed structurally into some domain-specific tasks while maintaining the common goals.

5.1.1 Application in the MAGIC-II Testbed

The monitoring system discussed in this section is particularly developed for the MAGIC-II (Multidimensional Applications and Gigabit Interconnect Consortium) project [31]. MAGIC-II is the second phase of a national ATM WAN sponsored by the Information Technology Office of the Defense Advanced Research Projects Agency. Figure 5.1

shows the graphical information of the topology and participants of the MAGIC-II project.

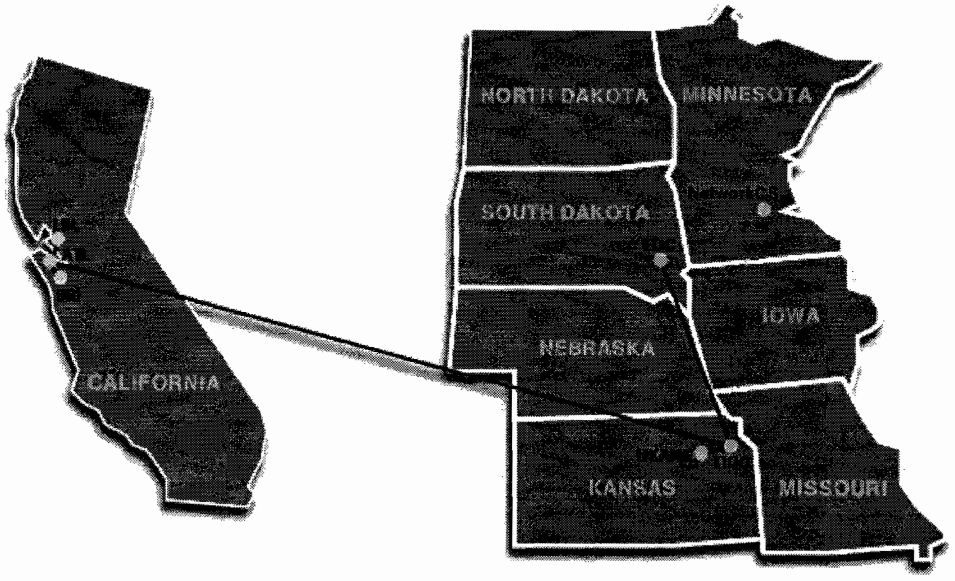


Figure 5.1: The MAGIC-II Network

The MAGIC testbed consists of three components:

- An interactive real-time, terrain visualization application, Terravision [30].
- A distributed parallel storage system (DPSS) [14] with performance sufficient to support the terrain visualization application
- A high-speed ATM network to link the computing resources required for real-time rendering of the terrain.

The monitoring system is intended to provide valuable information to dynamically configure the DPSS and to monitor the *health* of the MAGIC-II network in general. Particularly, the DPSS depends on the monitoring system to collect knowledge about some network characteristics to perform dynamic server selection, i.e. select a server from a number of servers which will give the *best* service. The network management aspect involves status and connectivity monitoring and network performance monitoring.

5.1.2 Objectives

The objectives of the work described in this chapter are twofold:

- To develop a monitoring agent which collects and maintains data about network state by doing testing and measurement on network elements
- To develop a tool that provides geographical display of application and network components and measurement data in a wide-area distributed application and network

5.2 Design Criteria for Monitoring System

The following design guidelines have been adopted to achieve the design objectives:

- Modularity:
  The monitoring agent framework must promote modular design which clearly separates the policy from the mechanism in doing network measurement.
- Portability:
  Since a distributed system most likely comprises heterogeneous components and systems, the agents should be easily portable and applicable to different architectures.
- Distributed:
  The system must be capable of monitoring network elements in more than one administrative domain.
- Extensibility:
  The capabilities of the agents in the system should be easily extended to support new types of measurement or testing.

5.3 Implementation Approach

NetSpec will be used as the main control entity of network testing and measurement because of its capabilities to perform distributed network testing in an integrated and extensible manner. This allows the system to be distributed and extensible at the same time. Moreover, since NetSpec has been ported to several major platforms, portability is not an issue.

For the agent implementation, we use the JATLite (Java Agent Template, Lite) [9] package which is written in Java language that allows users to quickly create new software agents that communicate robustly over the Internet. JATLite provides a basic infrastructure for agents' communication based upon TCP/IP and KQML messages. The use of the Java language allows the agents to be run on heterogeneous platforms and thus, ensures portability. Its modular construction consists of a hierarchy of increasingly specialized layers which may be customized to fit the specific requirements of a given system. Figure 5.2 shows how the hierarchy of layers are organized in JATLite.

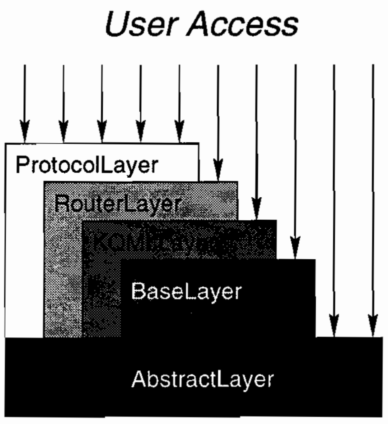


Figure 5.2: JATLite layers

The *Abstract Layer* provides the collection of abstract classes necessary for JATLite implementation. The Base Layer is built on top of the abstract layer and provides basic communication based on TCP/IP. The KQML Layer provides storage and parsing routines for KQML messages. The Router Layer provides name registration and message routing and queuing for agents. Finally, the Protocol Layer supports diverse standard internet services such as SMTP, FTP or HTTP.

One important concept in JATLite framework is the Agent Message Router (AMR) (also referred to as a *router*). It provides name registration and message routing or queuing for agents. In this scheme, agents can operate in disconnected mode and still receive the messages addressed for them. Another advantage is that the existence of an agent is transparent to the other agents in the system. An agent can send a message to another agent in the system by indicating the registered name of that agent in the destination field of the message and then sending the message to the router. The router then will forward the message to its intended recipient as long as it has registered itself with the router.

5.4 Functional Overview

This section provides the high level architecture of the monitoring system developed in the MAGIC-II testbed. In particular, it will define the different types of monitoring agents which exist in the system and how they relate to each other. The two main organizations involved in developing the distributed monitoring system are KU and LBNL. KU's contribution mainly involves monitoring at the network level while LBNL is particularly interested in monitoring at the application level. Both efforts are aimed at dynamic reconfiguration of the DPSS and also performance tuning and optimization of the distributed application.

Figure 5.3 shows the functional overview of the components existing in the system. The MAGIC-II testbed cloud represents the wide-area ATM network and also distributed application being monitored by the agents. There are four types of agents that we can identify from the picture:

NSAgent is a JATLite agent which creates and schedules NetSpec experiments and organizes the performance reports. The type and parameters of the experiment can be loaded dynamically. NSAgent collects the information about the DPSS system from the ServerMonitor.

VisAgent is a JATLite agent with a front-end applet which visualizes the state of the network and distributed application and the agents configuration. The information is collected from the NSAgent and ServerMonitor.

ServerMonitor is a JATLite agent which monitors the status and configuration of a DPSS system.

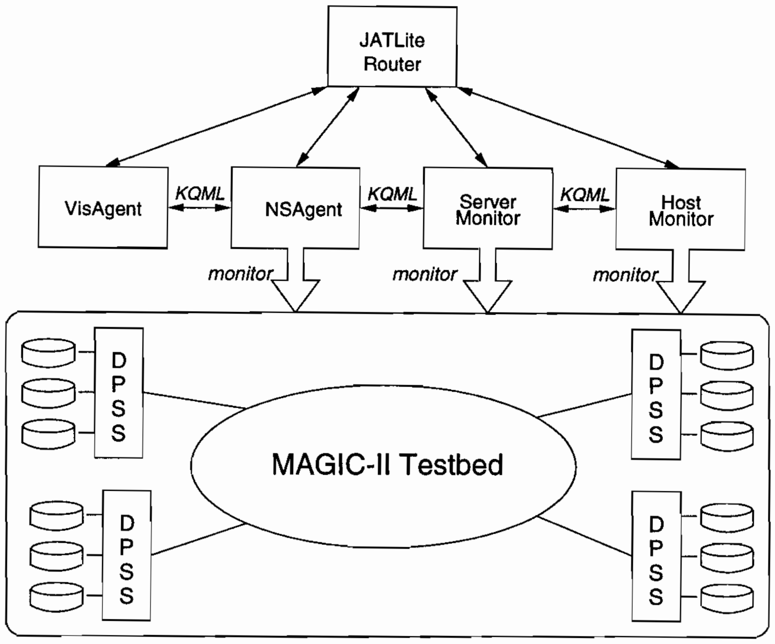


Figure 5.3: Functional Overview of Monitoring System

HostMonitor is also a JATLite agent which keeps track of the status of the currently connected DPSS clients.

Each of these agents registers itself with the router when it starts up. It exchanges KQML messages with other agents in the system via the router.

5.5 Implementation of NSAgent

The main responsibility of the NSAgent is to capture the state of some of the network characteristics and by doing so monitor the network. It does that by performing the appropriate test and measurement in the network. The result of monitoring and measuring the network characteristics can be used for different purposes. In the MAGIC-II testbed, the main objective is to use the knowledge about the current condition in the network to dynamically select the best server in the DPSS system. The characteristics of particular interest are the load of the network which can be represented by the available link bandwidth and round trip time. The NSAgent is also used to perform general network monitoring such as connectivity or throughput test.

Thus, the main focus of the NSAgent implementation is to design an *extensible* framework which can accommodate future types of network measurement or testing. Although the main objective in this project is to support the dynamic reconfiguration of the DPSS system, NSAgent should fulfill the ultimate goal of capturing the network state.

5.5.1 NSAgent Architecture

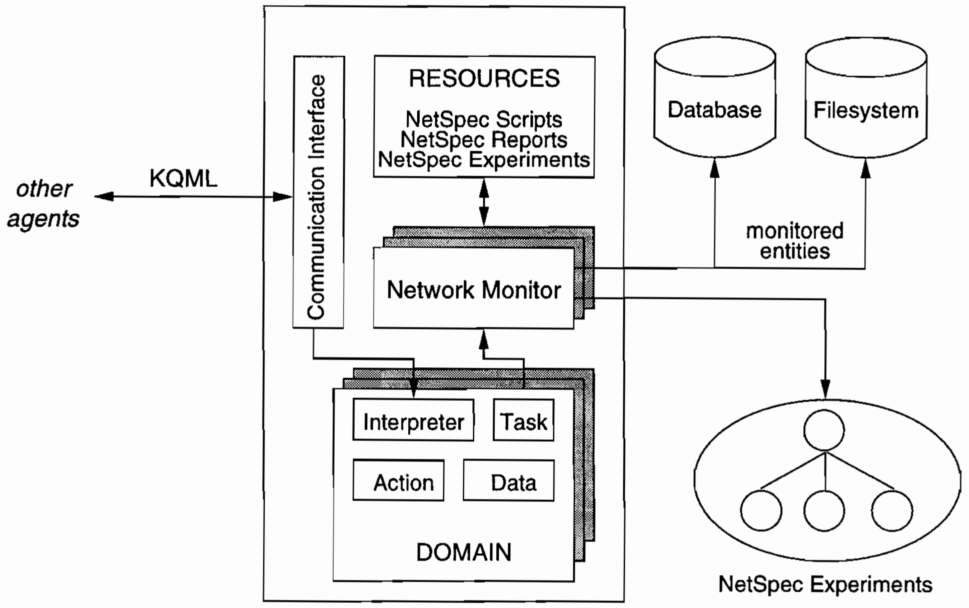


Figure 5.4: NSAgent Architecture and External Components

Figure 5.4 shows the architecture of the NSAgent. The JATLite's `RouterClientAction` class provides the basic communication interface based on TCP/IP socket via the router for receiving and sending KQML messages. The NSAgent has a collection of templates for NetSpec scripts and reports and for creating NetSpec experiments. It usually receives a monitoring task from other agents and creates the appropriate network monitor. The results of the NetSpec experiments can be stored in a database or the filesystem.

NSAgent also defines an abstraction called a *domain* which consists of an interpreter, an action, data structures and monitoring tasks. This abstraction allows NSAgent's capabilities to be extended dynamically. A domain can be loaded into NSAgent to handle a specific monitoring task at hand.

The message handling of a RouterClientAction agent is quite simple. When a KQML message destined for an agent arrives at the router, it stores the message in the incoming message box for that agent and then it will notify the agent. The agent is responsible for retrieving its own message and deleting them afterward. Program 5.1 shows the pseudo-code of the `Act()` method in the NSAgentAction class.

Program 5.1 Pseudo-code for the Act() method in NSAgentAction

```
class NSAgentAction extends RouterClientAction {
    ...
    public boolean Act (Object o) {
        // create KQMLmail
        KQMLmail mail = new KQMLmail ((String)0, 0);

        // extract KQMLmessage
        KQMLmessage kqml = mail.getKQMLmessage();

        // parse and interpret message
        ...

        // delete message
        addToDeleteBuffer(0);
      }
}
```

5.5.2 Knowledge Representation

One of the main components in a software agent is its knowledge base (KB). It provides the context of agent execution and the knowledge about its environment. The representation of knowledge can vary according to its purpose. NSAgent has two types of knowledge representation. The *procedural representation* uses program functions or methods to represent the data and the operation associated with an object. This type of representation is particularly useful in defining the capabilities of an agent. The second type of knowledge representation used in NSAgent is a *relational data base.* It usually serves as a back-end storage of static or run-time data. Relational data base provides a convenient and structured access and manipulation of data.

In NSAgent, the procedural representation is also called a *resource*. Section 5.5.2.1 will explain the types of resources which have been defined in NSAgent. Section 5.5.2.2 will describe the structure and configuration of the database.

5.5.2.1 Resources

NSAgent mainly uses resources to store the knowledge needed for running network experiments. Figure 5.5 shows the conceptual process of creating a network experiment. Given a task to monitor a specific network characteristic, NSAgent will create the appropriate network monitor object. The monitor object will create the NetSpec script that will invoke the proper NetSpec measurement daemon for the task. Then it will need to parse the performance report generated by the NetSpec daemon and interpret the result.

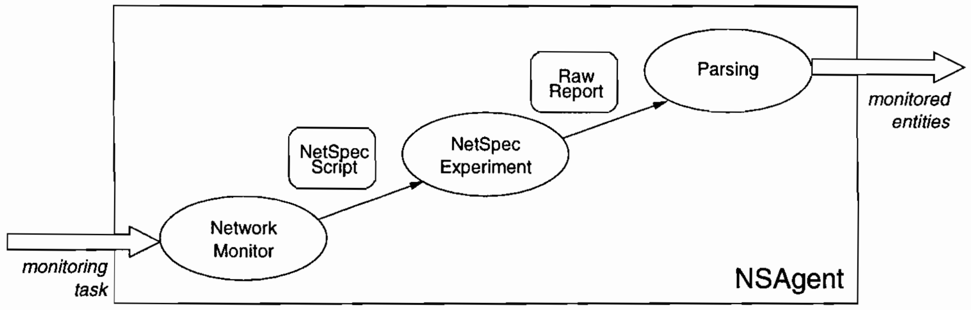


Figure 5.5: Network Monitoring Process

NSAgent defines the following resource abstractions:

- **Network Monitor**
  It represents the object that controls NetSpec experiment. Every network monitor object encodes the type and topology of NetSpec experiment. The type is associated with the kind of NetSpec measurement daemon which needs to be executed. The topology defines the execution construct and the participants of the experiment. A monitoring task must specify the type of monitor object to create. It can also specify the duration of the experiment and storage method.

For example, we can define a monitor object which performs a full mesh throughput experiment among N nodes. In this case, the type of the experiment (i.e. throughput) indicates that we have to use NetSpec Test Daemons. The full mesh topology will creates a script of ^{2} pair of end to end experiment between each two nodes in the set of N given nodes.

Table 5.1 gives some examples of the types of network monitors that have been implemented so far.

Name	Topology	Daemon type
FullMeshThroughput	N-to-N (full mesh)	*nstestd*
PointToMultiPointThroughput	1-to-N (star)	*nstestd*
EndToEndThroughput	1-to-1	*nstestd*
FullMeshDelay	N-to-N	*nspingd*
PointToMultiPointDelay	1-to-N	*nspingd*
EndToEndDelay	1-to-1	*nspingd*
HostMonitor	1-to-N	nssnmpd

Table 5.1: Network Monitors

- **NetSpec Script** A NetSpec script object contains the parameters and options of a NetSpec daemon and the methods to generate the script. A class must be defined for each NetSpec daemon since each NetSpec daemon has different parameters and format. For example, a NetSpec script class of the NetSpec Test Daemon defines the test parameters such as type of traffic, protocol and end-nodes. It also needs to implement the method that generates a block structure describing a this test.

- **NetSpec Report** Since there is no standard report format defined in NetSpec, each daemon generates varying types of report which makes parsing and interpretation of the report particularly difficult. To handle this problem, NSAgent defines a NetSpec report class for each type of daemon. Each class provides methods to parse and store the result.

5.5.2.2 Database

For this project, we have decided to use the msql (mini SQL) [20], a light-weight relational database based on SQL as the back-end storage. The NSAgent uses the database to store static configuration of the network (such as names and addresses of end hosts or switches) and the configuration of the distributed application being monitored, in this case, the DPSS. The results of network experiments can also be optionally stored in the database for further retrieval by other agents. Since the goal is to capture the state of the network, only the most recent result is kept in the database. Historical results can optionally be cached in the filesystem.

5.5.3 Domain Oriented Monitoring

As mentioned in Section 5.1, the knowledge about some characteristics of a computer network obtained from some network monitoring efforts can be applicable to different areas of application. Therefore, the architecture of a monitoring agent must be generic enough to cater to a broad range of monitoring aspect. To achieve this, the knowledge representation and the action of the agent must be abstracted away from domain specific information and goals.

NSAgent defines a procedural abstraction for a *domain* that encodes specific information defining the behavior of the monitoring agent in a domain's environment. A domain can be viewed as a *client* who is using monitoring services offered by the NSAgent and tailoring those services to its requirements. For example, a wide area network needs a monitor to check network connectivity and collect statistics from network elements. This network has its own network topology, measurement strategy and performance report format. Another network probably has different topology and monitoring goals. However, the underlying mechanisms of performing network measurement and data collection are essentially the same.

Each domain has a set of attributes to define its monitoring requirements:

- **Monitoring Task**

  A task defines the type of network monitor, the frequency of data collection, storage method and other parameters pertaining to the network measurement.

- **Message Interpreter**

  The definition of a domain should also include an interpreter for KQML messages related to this particular domain. The `ontology` field in a KQML message is used by the NSAgent to determine how to interpret this message. There is a one-to-one mapping between an ontology and a domain. NSAgent will pass the message for ontology A into the interpreter for domain A. This eliminates the need to create a separate agent to handle each type of monitoring domain existing in the system.

- **Initial Action**

  A domain also defines a set of initial actions to be taken after the domain definition is loaded into the NSAgent. Typical action includes database configuration and creation of monitoring tasks.

5.5.4 Examples of Domain Implementation

This section will give two examples of domain implementation in the context of MAGIC-II testbed. These two domains differ in the goals of monitoring. The DPSS domain mainly uses the monitoring information for dynamic reconfiguration of distributed application. The WAN monitor domain collects status information and perform active measurements to test a wide area network like the MAGIC-II testbed.

5.5.4.1 DPSS Domain

Goal

The objective of network monitoring in this domain is to obtain knowledge about the network link between a DPSS client and some DPSS servers. A client will select the server with the best connection (high throughput, low delay). This scheme is also called the dynamic server selection problem and has shown promising results [6].

Monitoring Tasks

Given the name and location of the DPSS servers in the system, NSAgent must perform network measurements to test the link quality between a given set of DPSS clients and the servers. Specifically, NSAgent creates a throughput experiment and a delay experiment with star topology for every registered DPSS client.

Interpreter

The KQML messages in this domain mainly deal with the configuration of the DPSS system, for example to register a new DPSS client. Other agents in the DPSS system can also send query to find out about the current link condition between a client and a server.

Initial Action

When the DPSS domain is loaded, it will instruct NSAgent to find out more about the DPSS configuration. After the information is available, NSAgent can start the monitoring tasks described above.

5.5.4.2 An ATM Wide Area Network Monitor Domain

Goal

The objective is to collect statistics from network elements, monitor the connectivity in the network and perform periodic throughput measurement and some other general network monitoring activities.

Monitoring Tasks

Several monitoring tasks can be defined:

- Full mesh connectivity test among all sites in the network
- Full mesh throughput test among all sites
- Status of network elements and end hosts in each site

Interpreter
The interpreter for this domain can handle queries about the status of network monitors and a customized performance report. There should also be provisions for adding a new set of monitoring capabilities or removing existing monitors.

Initial Action
The information about each network elements and network configuration is loaded. Basically, an ATM wide area network can be hierarchically organized into a collection of sites connected by ATM switches. Each site consists of a collection of end hosts and switches.

5.6 VisAgent Implementation

VisAgent is a front-end interface to the monitoring systems described in this chapter. It aims at providing geographical display of the state of a distributed application and its underlying wide-area network. The logical approach is to make the visualization tool itself an agent that communicates with the monitoring agents and hence, the name VisAgent.

VisAgent is implemented as a Java applet which can be loaded from a web browser or launched as a Java application. This approach provides the convenient access for a thin client to access the visualization tool from various location or environment. JATLite's router mechanism plays an important role especially for this type of agent because the only information needed by the VisAgent to communicate with the rest of the system is the address of the router. It also solves the security problem imposed by web browser on Java applet which only permits an applet to create sockets to processes on the same host where the web server resides. As long as the JATLite's router and the web server are configured to run on the same hosts, the applet can be loaded from anywhere.

VisAgent uses both polling and event-driven mechanism in updating the display. Polling is mainly used to collect information which are supplied by other agents, while event driven is used to collect information from the database. This strategy is used to achieve the reactiveness of the visualization tool. Query to an agent usually takes significantly more time than query to a database. Therefore, user's actions usually only trigger query to a database and an update of the view.

5.6.1 GenMap Package

The main goal of VisAgent is to display information based on their geographical position. Therefore, we need a user interface with a map overlaid with visual symbols to represent state of the system. The GenMap package [17] provides a starting point to achieve this goal. The package consists of a set of Java classes which provides the basic functionality for geographical network visualization. It implements the classes to draw the background map, nodes and lines and methods to zoom in and out the map.

Considerable amount of effort has been devoted to adapt the GenMap package to do the type of visualization that is required for MAGIC-II. GenMap is really specific as to the format and size of background map used. The original package uses a flat map of the whole world which then can be zoomed in to a particular continent. If the resolution of the base map is not good enough, the zoomed version of a continent will be of very poor quality. Since this project is particularly interested in providing visualization for the United States region, we want to start with a US map as the base map. The solution is to modify the ImageProducer class which supplies the pixels to be drawn on the screen so that we can start with a map of a particular region bounded by a rectangle of the given latitude-longitude pairs.

Another addition to the GenMap package is the thumbnail map which shows a rectangle bounding the current display on the base map. This feature is particularly useful if the base map does not have details such as state lines or city names. Users can always refer to the thumbnail image to figure out which part of the map they are looking at.

5.6.2 Visualization Layers

The VisAgent collects information from various sources and tries to aggregate them to form a unified view of a distributed application and its underlying wide-area network.

The best way to organize the data is to group them by the source of information. Visually, we can provide the display as viewed from a specific layer. By separating the information in layers, we can potentially display many characteristics of the application and the network in one convenient visualization tool.

VisAgent provides three layers of visualization. The first layer is for the distributed application, the second for the network and the third is for the monitoring agents. The next few paragraphs will describe each layer in detail and shows the screen shot.

Application layer shows the location of the components of the distributed application and the status of each component. In the MAGIC-II context, each node represents either a DPSS master, server or client. The lines represent the active connection from a client to the server(s). This layer provides the information about the number of servers, the location and configuration of each and identifies the location and status of the client.

Network layer shows the configuration of the network and the results of the measurement done on the network. Each node represents a site in the testbed. Lines represent the physical connection or the network characteristics. The width of the lines usually reflects the value of the network characteristics they represent. This layer also provides the detail configuration of network elements (e.g. switch) and end-hosts in each site.

Agent layer shows the configuration of the monitoring system, i.e. the geographical location and the address of the agents. It also shows the topology and status of the active network experiments. This layer can provide useful information to understand the components and interaction between elements in the monitoring system.

5.6.3 Visual Element Mapping

For each visualization layer, each node and line can represent different entity and value. A node representation can be varied in terms of shape, size and color. Line can only vary in size and color. The mapping of the attributes of the nodes and lines to measured parameters can either be hard coded in the program or dynamically reconfigurable at run time. To make our tool as general and flexible as possible, we choose the second approach.

For each layer we define a set of mappings for the nodes and lines and provide an efficient API for the programmer. The GenMap's base class for node and line have been modified so that the attribute binding is done as late as possible, for example just before the node and line are displayed on the screen. The mapping can be defined in a configuration file which is loaded initially. The attributes of a node or a line are assigned by consulting the rules defined in the configuration. The configuration is also used to create legend for each visualization layer that is updated every time the view changes to another layer.

For each layer, two lines describing the node and line's attributes, respectively, must be specified in the configuration file. Table 5.2 summarizes the attributes for the node and line and acceptable values for each attribute.

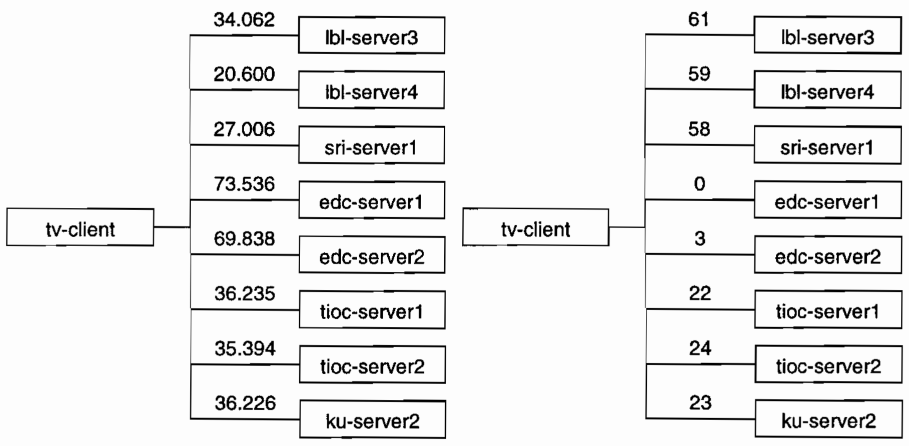
Element	Attribute	Value Syntax	Description
Node/ line	LABEL	*Name*	The type of entity this node/line represents
	UNIT	*Name*	The unit for the value represented by this node/line
	COLOR	FIX(*color*	The hex value of the color for all nodes/lines
		RANGE(*min,max*)	The color of this node/line can vary according to the value which lies between *min* and *max*
		LIST((*label1, color1*), (*label2, color2*) ...)	Node/line with *label1* is colored *color1*, and so on
	SIZE	FIX(*size*	All nodes/lines have the same size
		RANGE(*min,max*)	The size of this node/line can vary according to the value which lies between *min* and *max*
		LIST((*label1, size1*), (*label2, size2*) ...)	Node/line with *label1* is of size *size1*, and so on
Node	SHAPE	FIX(*shape*)	All nodes have the same shape
		LIST((*label1, shape1*), (*label2, shape2*) ...)	Node/line with *label1* is of shape *shape1*, and so on

Table 5.2: Visual Element Mapping Attributes

Upon receiving this message, the NSAgent instantiates two types of network monitors (see Table 5.1:

1. A *PointToMultiPointThroughput* monitor measures the transfer capacity (throughput) of the links between the client and the servers. If either the client or the server has more than one network interface, each interface needs to be tested. This experiment is done once every 30 minutes. The results of the experiment is stored in the database. Figure 5.7(a) shows the variation in the throughput from tv-client to the eight servers in the demonstration as reflected in the database at one point in time.

2. A *PointToMultiPointDelay* monitor measures the round trip time between the client and the servers. Since this type of experiment does not produce a great disturbance to the network, it can be done more frequently (once every 15 minutes) without consuming too much network resources. An example result of the measurement is shown in Figure 5.7(b).



(a) Throughput (in Mbps) (b) RTT (in msec)

Figure 5.7: Variation in throughput and round trip time between a DPSS client and servers

5.7.1.2 Monitoring Connectivity in the Network

The objective of this task is to monitor the connectivity in the network. Sometimes it is difficult to identify the cause of the problem when an 'Unreachable destination' message is received because of the network has many potential points of failure. This is especially true in an experimental testbed such as MAGIC-II where the configuration of the network may change frequently. Many times we found that the reachable machines/sites varied greatly from one machine to the next in the network.

The objective of the measurement activity described in this section is to provide a better understanding about how the sites in the network are connected to each other and what kind of connectivity exists currently. Every 15 minutes, NSAgent schedules a *FullMeshDelay* experiment among the major hosts in each site in the network. Since *FullMeshDelay* network monitor uses the ICMP_ECHO mechanism to measure the round trip time, it can be utilized to test the connectivity from one point it the network to several other points.

5.7.1.3 Monitoring Transfer Capacity of the Network

Besides monitoring the connectivity, another important metrics in assessing the general health of a network is to test the transfer capacity (throughput) of the links in the network. Variation in the throughput is generally affected by the amount of traffic and the presence of bottleneck links in the network. For this purpose, NSAgent creates a *FullMeshThroughput* experiment among the major hosts in each site. Since this type of experiment introduces a large amount of test traffic to the network, it should not be done too frequently. However, to study the variation in the throughput during the course of a day, it should be done at different times during the day. In the demonstration, NSAgent schedules this experiment once every 3 hours.

5.7.1.4 Monitoring Network Elements Status

While the measurement activity described in Section 5.7.1.2 is aimed at providing information about connectivity between sites in the network, it does not provide detail information about each machine or other network element within a site. This type of

5.7 Example Configuration in MAGIC-II Testbed

This section will describe the configuration and capabilities of the monitoring system described above demonstrated during the MAGIC-II quarterly meeting on July 14, 1998 at the University of Kansas. Figure 5.6 shows the configuration of the demo.

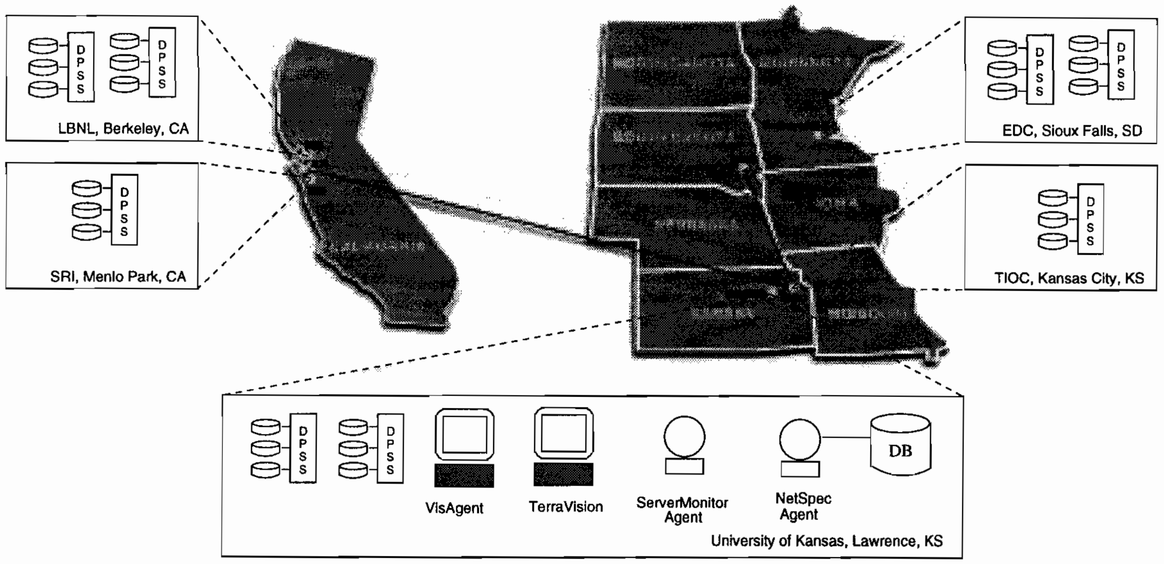


Figure 5.6: Demonstration Configuration

We had 8 DPSS servers distributed across the MAGIC sites. The agents, the JATLite router, the database server and the web server were running on a machine at KU (*faraday.ukans.magic.net*, a Sun Ultra Sparc running Solaris 2.6). NetSpec was installed at all DPSS hosts and at least in one machine at each site. The domain oriented monitoring described in section 5.5.3 was implemented after the demonstration date to make the monitoring system more applicable to areas other than the DPSS. Prior to that, the configuration of NSAgent described in this section contained specific information about the DPSS (for the description of the previous version, see the document in [48]. Adapting the configuration described in the following sections to the new framework requires only minimal changes which will be noted clearly in each section.

5.7.1 NSAgent Configuration

For the demonstration, NSAgent was configured to schedule and run a number of network experiments by using NetSpec. Each network measurement was encapsulated in a monitoring task presented to the NSAgent during its initialization. By default, NSAgent loads a series of KQML messages from an initialization file which defines the agent's initial behavior. Appendix A describes the KQML messages implemented by the NSAgent.

The following subsections will describe each monitoring task and its significance as well as some example results obtained from the measurement when appropriate.

5.7.1.1 Monitoring Link Quality between DPSS Client and Servers

The objective of this activity is to determine the link quality between a DPSS client and the servers configured in the system in order to select the server with the best connectivity to the client. The link quality is defined in term of round trip delay and transfer capacity of the link.

When a new DPSS client comes up in the system, the NSAgent must be notified about its existence (please refer to the KQML message A.1 which performs this in Appendix A). The entity that must register the client to the NSAgent can be the client itself or a DPSS monitor agent which continually keeps track of the emergence of a new client. Since this had not been implemented by the demonstration date, a static configuration was used instead. The following KQML message was included in the initialization file to register a DPSS client named `tv-client` with only one network interface, *terravision.ukans.magic.net*, which will potentially talks to one of the eight DPSS servers.

```
(evaluate :sender NSAgent :receiver NSAgent
   :content (tell-resource :type client
               :name ku-client1
               :pos (38.963 -95.233)
               :interface (terravision.ukans.magic.net 198.207.143.157)
               :server (lbl-server4 sri-server1 tioc-server1 tioc-server2
                        lbl-server3 ku-server1 edc-server1 edc-server2)))
```

monitoring can be achieved by doing a *PointToMultiPointDelay* between a host in a site to the remaining hosts and switches in each site. The database contains the static information about the configuration in each site. NSAgent uses this information to create experiment which sends ICMP_ECHO message every 15 minutes from a designated host to the rest of the network element in each site.

5.7.1.5 Adapting to Domain Oriented Monitoring

In the new framework which uses domain oriented monitoring, the monitoring tasks are encapsulated in a domain module. The measurements related to the DPSS described in Section 5.7.1.1 is in one domain called `DPSSDomain` and the measurements in Sections 5.7.1.2, 5.7.1.3 and 5.7.1.4 are encoded in a separate domain, `NetMonDomain`.

Instead of loading the network monitors during the NSAgent initialization, both of these domains are loaded instead. From that point on, NSAgent works in the same way as in the old approach with a freedom to load or unload domains during runtime.

5.7.2 VisAgent Configuration

In the demonstration, the VisAgent is started by loading the Java applet for the visualization from a Java capable browser. VisAgent collects the performance data and static configuration from three sources: the NSAgent, the database server and the Server-Monitor agent. KQML messages are used in communicating with the agents, while standard SQL messages are used to interact with the database server. Appendix B describes the KQML messages which are used by the VisAgent to interact to the other agents in the system. As described in Section 5.6.2, VisAgent provides three layered views:

1. **Application Layer**
   Color-coded nodes represent the DPSS master, server and clients. The placement of the nodes indicates the geographical location of the corresponding DPSS components. Lines represent the *active* connection between a client and some DPSS servers. Figure 5.8 shows the screen capture of the visualization tool displaying the status of the DPSS. In this figure, a DPSS client, `ku_client1`, is accessing

data set from two DPSS servers as shown by the lines connecting the client and the servers. The data panel at the bottom left corner shows the detail information about `ku_client1` such as its network interfaces and the results of the throughput measurements. The panel on the right side displays the names of the DPSS master(s), server(s), and client(s) that are currently registered with the monitoring systems.

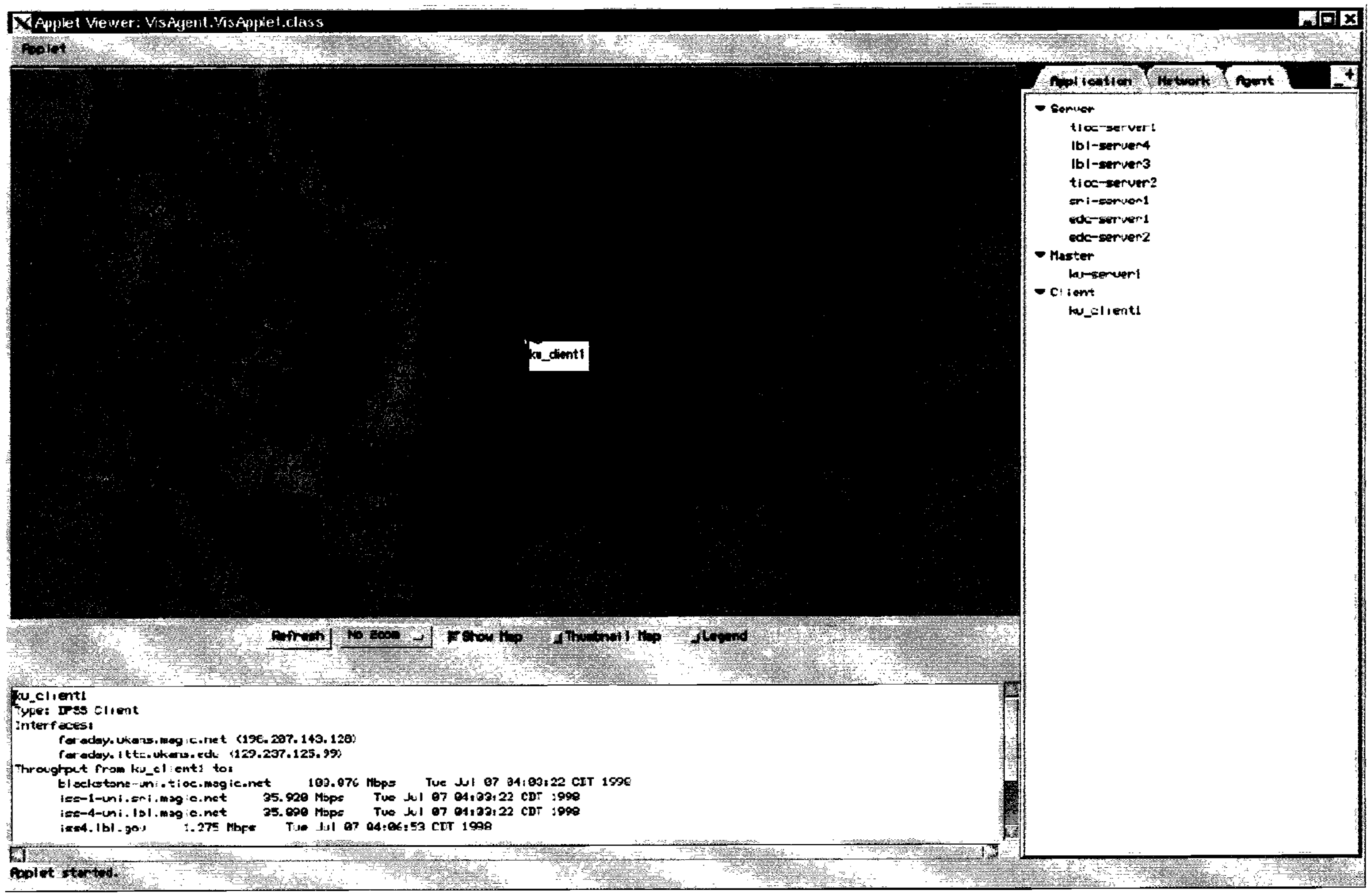
Actions on this layers include:

- Clicking on a DPSS master or server's node will send a *GetHostInformation* query about that node to the ServerMonitor agent.
- Clicking on a client's node will send a *LinkInformation* query to the NSAgent about the latest throughput and delay numbers from that client to the servers.
- VisAgent periodically polls the ServerMonitor to collect the information about the currently connected client by sending the *GetMasterInformation* query. The information typically includes the user name, the program name and the data sets accessed by the client. When a new user is detected, the view is updated with lines representing connection from the user to the corresponding DPSS servers.

2. **Network Layer**

   Each node at the network layer represents a site in the MAGIC-II testbed. The network layer is further divided into three sublayers:

   - The topology sublayer shows the physical configuration of the testbed. The color of the lines represent the physical link's capacity (i.e. DS-3, OC-3, OC-12). Clicking on a node will bring up another window which shows the detail configuration and status of the network elements in each site (see Section 5.7.1.4. Elements which are down are colored differently from elements which are up. Figure 5.9 shows the screenshot of the VisAgent displaying the topology sublayer. The smaller window shows the detail configuration at KU site. The panel on the right displays the names of the network elements and hosts at each site. Green color is used for network elements and



```
ku_client1
Type: DPSS Client
Interfaces:
      faraday.ukans.magic.net (196.207.143.128)
      faraday.ittc.ukans.edu (129.237.125.99)
Throughput from ku_client1 to:
      blackstone-uni.tioc.magic.net     103.076 Mbps    Tue Jul 07 04:03:22 CDT 1998
      iss-1-uni.sri.magic.net     35.920 Mbps    Tue Jul 07 04:03:22 CDT 1998
      iss-4-uni.lbl.magic.net     35.090 Mbps    Tue Jul 07 04:03:22 CDT 1998
      iss4.lbl.gov     1.275 Mbps    Tue Jul 07 04:06:53 CDT 1998
```

Figure 5.8: Screenshot of VisAgent at the Application Layer. Nodes represent DPSS master, servers and clients. Lines represent active connection between a DPSS client and servers.

blue color is used for hosts. Any element or host which is not responding to a PING message is indicated by a red color.

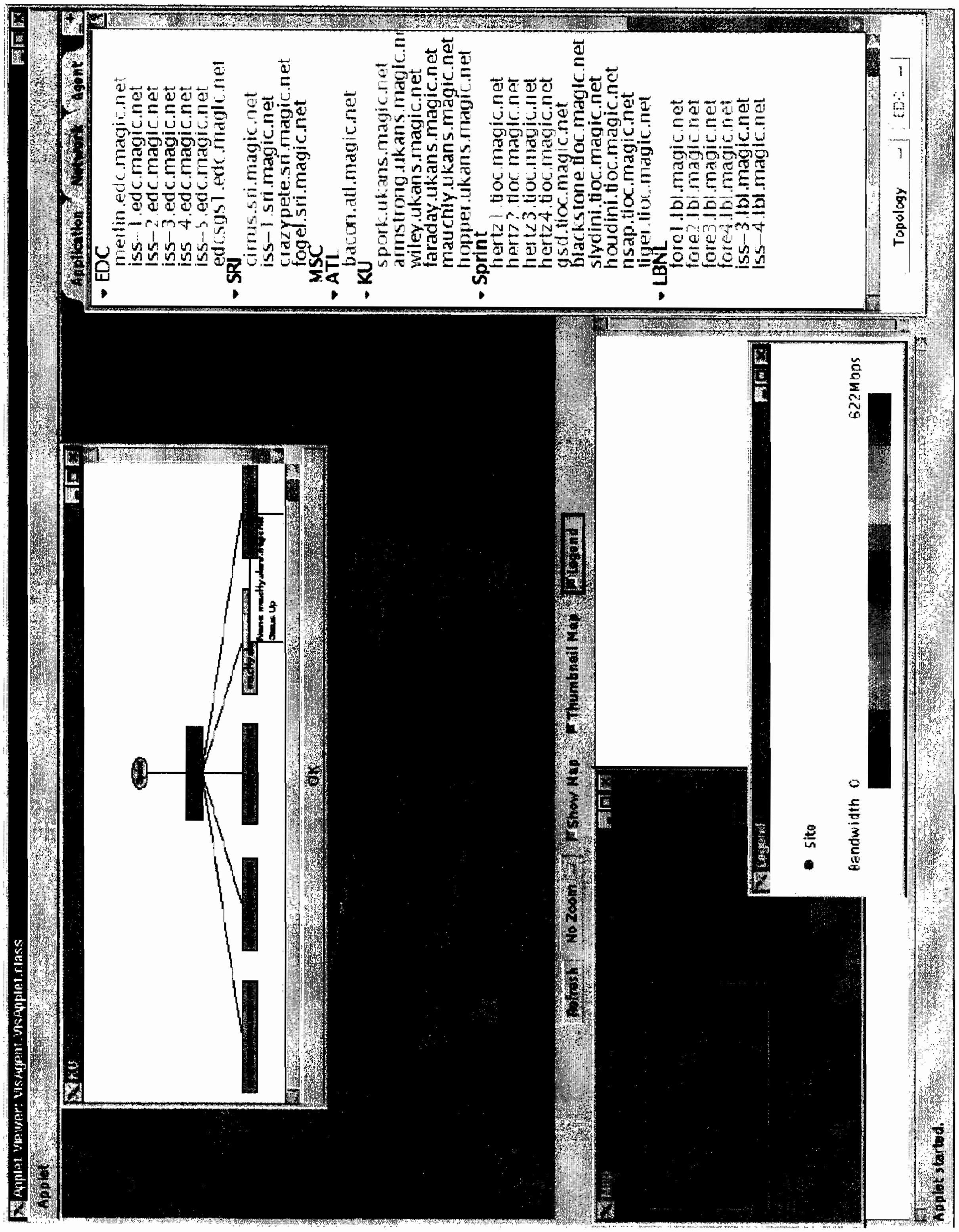
- The connectivity sublayer shows the result of the connectivity test (see Section 5.7.1.2 from a point of view of a site. The color of the lines coming out from a site to another site represents the round-trip-time in millisecond. Unreachable site will not be connected by a line. Figure 5.10 shows the screenshot of the VisAgent displaying the connectivity as seen from EDC site. The color of the lines represent the round-trip time values as indicated by the color spectrum in the legend window. The round-trip time information is also displayed in the data panel.
- The maximum bandwidth sublayer shows the result of the throughput test from a point of view of a site (see Section 5.7.1.3). The color of the lines represents the achievable throughput in Mbps.

3. **Agent Layer**
   Each node in this layer represents either a JATLite agent or a NetSpec daemon. The color of the node indicates the types of agent or NetSpec daemon. Lines connecting NetSpec daemon's nodes indicate the topology of the experiment. The view of the agent display is automatically updated when a new experiment begins or an existing experiment terminates. For example, figure 5.11 shows the visualization at the agent layer when a full mesh delay experiment is taking place. The blue nodes represent the NetSpec Ping Daemons involved in the experiment. The lines between the nodes represent the topology of the experiment.

The script used to configure the visual element mapping is given in Appendix C.

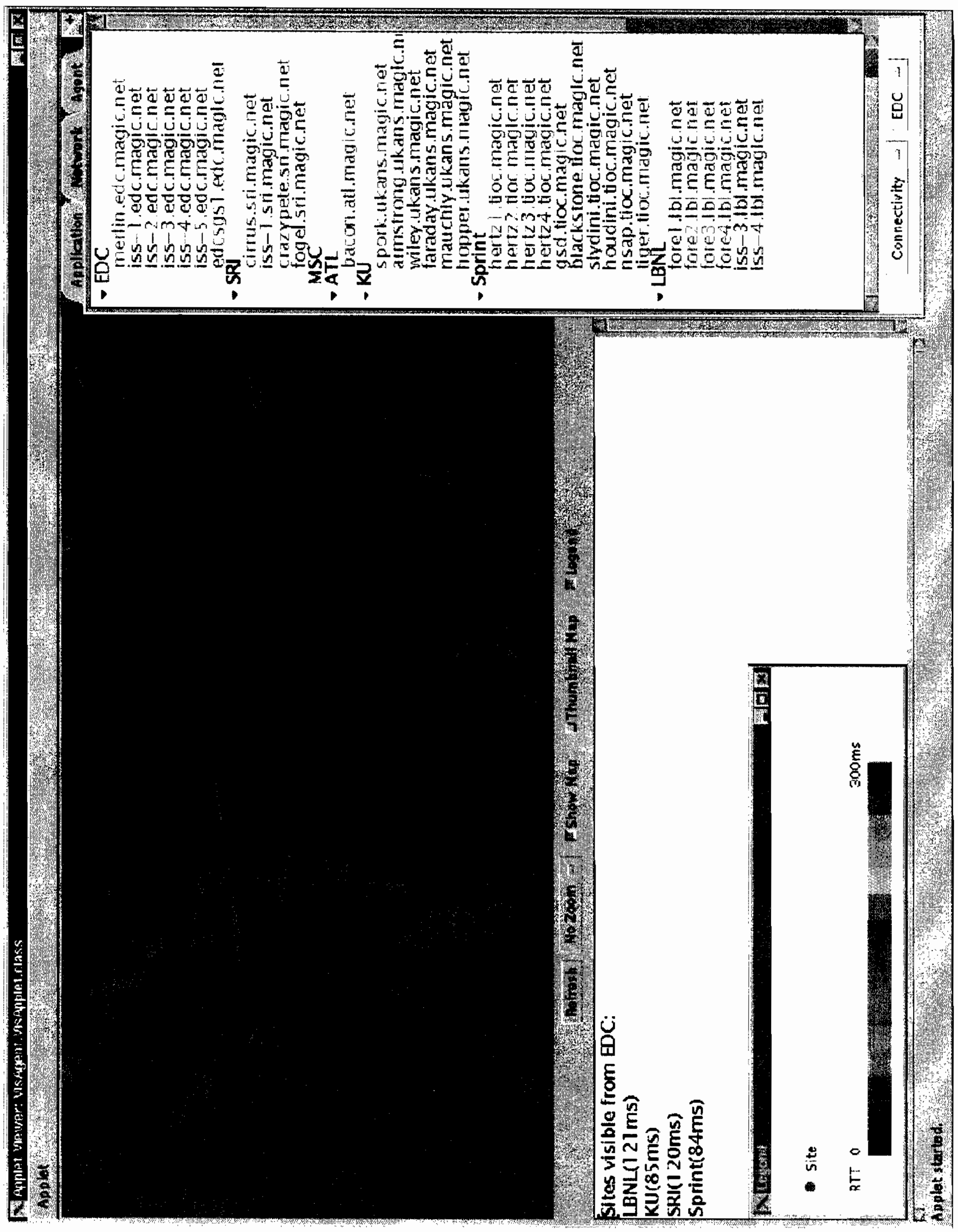
5.8 Summary

This chapter has discussed the design and implementation of the monitoring system developed in the MAGIC-II network. The system consists of monitoring agents which communicate with each other using KQML. NSAgent schedules network experiment and organizes the performance data. VisAgent aggregates data collected by the agents



- EDC
  - merlin.edc.magic.net
  - iss-1.edc.magic.net
  - iss-2.edc.magic.net
  - iss-3.edc.magic.net
  - iss-4.edc.magic.net
  - iss-5.edc.magic.net
  - edcsgs1.edc.magic.net
- SRI
  - cirrus.sri.magic.net
  - iss-1.sri.magic.net
  - crazypete.sri.magic.net
  - fogel.sri.magic.net
- MSC
- ATL
  - bacon.atl.magic.net
- KU
  - spork.ukans.magic.net
  - armstrong.ukans.magic.n
  - wiley.ukans.magic.net
  - faraday.ukans.magic.net
  - mauchly.ukans.magic.net
  - hopper.ukans.magic.net
- Sprint
  - hertz1.tioc.magic.net
  - hertz2.tioc.magic.net
  - hertz3.tioc.magic.net
  - hertz4.tioc.magic.net
  - gsd.tioc.magic.net
  - blackstone.tioc.magic.net
  - slydini.tioc.magic.net
  - houdini.tioc.magic.net
  - nsap.tioc.magic.net
  - liger.tioc.magic.net
- LBNL
  - fore1.lbl.magic.net
  - fore2.lbl.magic.net
  - fore3.lbl.magic.net
  - fore4.lbl.magic.net
  - iss-3.lbl.magic.net
  - iss-4.lbl.magic.net

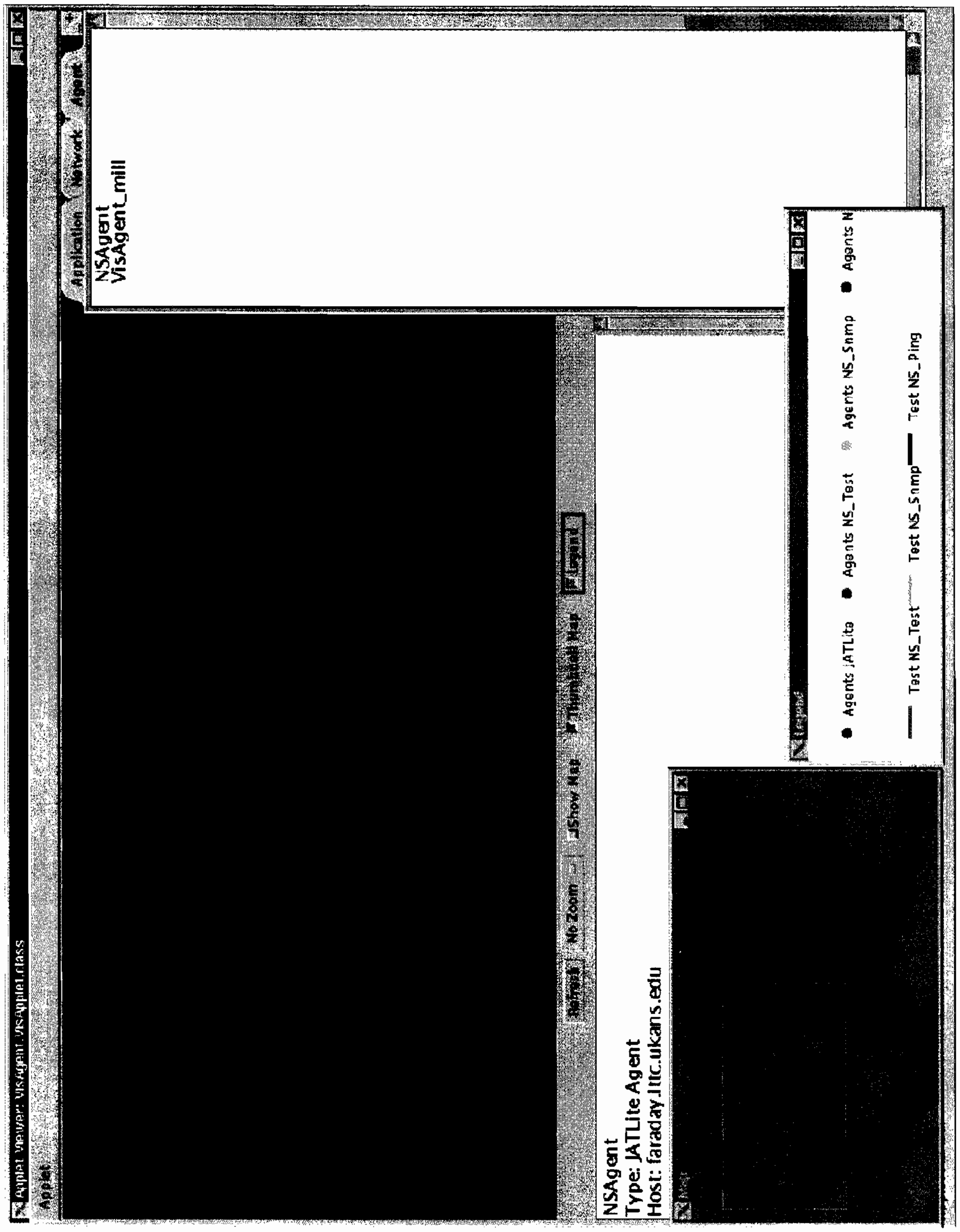
Figure 5.9: Screenshot of VisAgent at the Network Topology Sublayer. Nodes represent sites (organizations). Lines represent the bandwidth of the physical connection between sites.



```
▾ EDC
    merlin.edc.magic.net
    iss-1.edc.magic.net
    iss-2.edc.magic.net
    iss-3.edc.magic.net
    iss-4.edc.magic.net
    iss-5.edc.magic.net
    edcsgs1.edc.magic.net
▾ SRI
    cirrus.sri.magic.net
    iss-1.sri.magic.net
    crazypete.sri.magic.net
    fogel.sri.magic.net
  MSC
▾ ATL
    bacon.atl.magic.net
▾ KU
    spork.ukans.magic.net
    armstrong.ukans.magic.ne
    wiley.ukans.magic.net
    faraday.ukans.magic.net
    mauchly.ukans.magic.net
    hopper.ukans.magic.net
▾ Sprint
    hertz1.tioc.magic.net
    hertz2.tioc.magic.net
    hertz3.tioc.magic.net
    hertz4.tioc.magic.net
    gsd.tioc.magic.net
    blackstone.tioc.magic.net
    slydini.tioc.magic.net
    houdini.tioc.magic.net
    nsap.tioc.magic.net
    liger.tioc.magic.net
▾ LBNL
    fore1.lbl.magic.net
    fore2.lbl.magic.net
    fore3.lbl.magic.net
    fore4.lbl.magic.net
    iss-3.lbl.magic.net
    iss-4.lbl.magic.net
```

```
Sites visible from EDC:
LBNL(121ms)
KU(85ms)
SRI(120ms)
Sprint(84ms)
```

Figure 5.10: Screenshot of VisAgent at the Network Connectivity Sublayer. Nodes represent sites (organizations). Lines represent existing connectivity and their color represent the round trip time between sites.



NSAgent
Type: JATLite Agent
Host: faraday.ittc.ukans.edu

Figure 5.11: Screenshot of VisAgent at the Agent Layer. Nodes represent JATLite agents or NetSpec daemons. Lines connecting NetSpec daemon nodes represent NetSpec experiment's topology.

and provides the visualization for performance data in three layered views. The configuration and example agents demonstrated during the MAGIC-II quarterly meeting has been described.

Chapter 6

Conclusions and Future Work

The rapid growth of computer networks has made the process of understanding the interaction among network components more challenging than ever. The increase in the size of the network is accompanied by more demanding use of the network by distributed applications that critically rely on the network to function well. Consequently, monitoring the health and stability of the network has become crucial.

Numerous efforts have been devoted to measure the performance of the network for the purpose of network management or performance evaluation through creation of measurement tools or network probes. However, there is yet a standardized "measurement infrastructure" which offers the systematic control and management of measurement efforts and performance data. The Internet is a classic example of an under-measured and under-instrumented network. As a result, the quality of service over the Internet has dramatically worsened in the past years, creating an *Internet gridlock* [12].

The work described in this thesis has addressed the issues of creating a scalable and extensible network measurement infrastructure. Because of its extensible and flexible architecture, NetSpec has been used as the main control structure to create measurement daemons in the network. An important extension to support continuous collection of performance data has been added to the NetSpec framework. The extension allows NetSpec to be used in a more real-time environment and for collecting data with a finer granularity.

Two examples of the implementation of monitoring daemons that utilize the continuous monitoring feature have been presented: (1) the SNMP daemon collects performance data from network devices by using SNMP and (2) the Data Stream daemon collects data from an operating system kernel through the Data Stream Kernel Interface.

This thesis also describes the implementation of a monitoring agent system as an approach to creating a network measurement infrastructure. The role of the agents in the system is to *automate* the process of (1) the continuous monitoring of application components and network characteristics, (2) the creation and control of network testing and measurement, (3) the collection and storage of performance data, and (4) the correlation and presentation of performance data to application and/of network manager.

In the MAGIC-II testbed where this monitoring system was developed and tested, its main purpose is to capture some characteristics of the network state to support dynamic reconfiguration of the DPSS. In addition, the monitoring system is also used to monitor the general health of the MAGIC-II network by measuring some vital metrics of the network, such as connectivity, latency and throughput. The monitoring system also includes a visualization tool that serves as a front end interface to the performance data and more importantly, provides an integrated view of distributed application and the underlying network.

6.1 Future Work

The work of creating the ultimate network measurement infrastructure is an ongoing research. The approach described in this report provides several opportunities for further work:

- Extend the NetSpec protocol to allow disconnected operation of NetSpec daemon. This feature can be useful for long term monitoring in which it is not desirable for the NetSpec controller to linger for a long period of time. The collected data can be sent through the report channel to a stand-by report collector server.

- The architecture of the monitoring agent described in Chapter 5 can be improved by including rule-based approach in creating network experiments as done in [33]. In such a scheme, a network monitoring action is defined by a set of rules. Instead of programming monitoring tasks for each domain, a script for the rules can be specified.

- The implementation of network monitors can be simplified if the topology information is separated from daemon-specific parameters/information. Since the topology of an experiment is usually applicable to different type of NetSpec daemons, most of the topological abstraction can be reused.
- The organization of configuration and historical data should be improved and standardized by creating a more structured data base for storing performance data.

Appendix A

KQML Messages Implemented by NSAgent

A.1 Registering a New DPSS Client

```
(evaluate :sender xxx :receiver NSAgent
          :content (tell-resource :type client :name clientName
                                  :pos (lat long)
                                  :interface (iface1 ipaddr1 iface2 ipaddr2 ...)
                                  :server (server1 server2  ...)))
```

Description:

Register a new DPSS client. NSAgent will monitor the links connecting to the given set of servers for each network interfaces and store the information in the database.

A.2 Unregistering a DPSS Client

```
(evaluate :sender xxx :receiver NSAgent
          :content (invalidate-resource :type client :name clientName))
```

Description:

Unregister a DPSS client. NSAgent will stop monitoring the links connected to this client and remove information from the database.

A.3 Getting Performance Measurement Result on a Link

```
(ask-one :sender xxx :receiver NSAgent
         :content (GetLinkInformation ClientName client ServerName server))
```

Description:

Get the throughput and delay values for the link(s) between client and server.

```
(reply :sender NSAgent :receiver xxx
       :content (LinkInformation ClientName client ServerName server
                                 (Interface iface1 Throughput t1 Delay d1)
                                 (Interface iface2 Throughput t2 Delay d2) ...))
```

Description:

Reply for the GetLinkInformation query. Throughput and delay values for every network interface are returned.

A.4 Getting the Names of Configured DPSS

```
* (ask-one :sender xxx :receiver NSAgent
           :content (GetKnownDPSS)

{\bf Description:} \\
Get the names of all DPSS servers and clients known to NSAgent.

\begin{singlespace}
\begin{small}
\begin{verbatim}
(reply :sender NSAgent :receiver xxx
       :content (KnownDPSS ClientNames client1 client2 ...
                           ServerNames server1 server2...))
```

Description:

Return the names of DPSS clients and servers known to NSAgent.

A.5 Getting the Names of Known DPSS Clients

```
(ask-one :sender xxx :receiver NSAgent
          :content (GetKnownClients)

{\bf Description:} \\
Get the names and addresses of all DPSS clients known to NSAgent.

\begin{singlespace}
\begin{small}
\begin{verbatim}
(reply :sender NSAgent :receiver xxx
       :content (KnownClients (ClientName client1 Interfaces iface1 ...)
                              (ClientName client2 Interfaces iface1 ...)
```

Description:

Returns the names and addresses of all DPSS clients known.

A.6 Getting Detail Information on all Registered DPSS Clients

```
(ask-about :sender VisAgent :receiver NSAgent
           :content (dpss-clients))
```

Description:

Get the information about the DPSS clients registered with NSAgent: name, position, servers.

```
(reply :sender NSAgent :receiver xxx
       :content (dpss-clients :value ('(name lat long (server1 server2 ...))
                                      '(name lat long (server1 server2 ...))) ))
```

Description:

Returns the information about registered clients.

A.7 Getting Information about Active NetSpec Experiments

```
(ask-about :sender VisAgent :receiver NSAgent
           :content (netspec-experiments) )
```

Description:

Get the information about active NetSpec experiments.

```
(netspec-experiments :sender NSAgent :receiver xxx
                     :content ( (NetMonName NetmonType NumHosts host1 host2 ... )
                                (NetMonName NetmonType NumHosts host1 host2 ... )
```

Description:

Returns the information about active NetSpec experiments.

A.8 Getting the Status of Network Monitors

```
(ask-about :sender VisAgent :receiver NSAgent
           :content (network-monitors) )
```

Get the information about the status of network monitors created by NSAgent.

```
(netspec-experiments :sender NSAgent :receiver VisAgent
                     :content ( (NetmonName status)
                                (NetmonName status) ... )
```

Description:

Returns the names and status (idle or running) of network monitors.

A.9 Registering a VisAgent

```
(register-visagent :sender VisAgent :receiver NSAgent
                   :name VisAgentName )
```

Description:

Register a VisAgent with NSAgent. A registered VisAgent will get updates about experiment results and status.

A.10 Unregistering a VisAgent

```
(unregister-visagent :sender VisAgent :receiver NSAgent
                     :name VisAgentName )
```

Description:

Unregister a VisAgent.

A.11 Creating a Network Monitor

```
(create-experiment :sender xxx :receiver NSAgent
                   :type NetMonType
                   :param (NetMonName :key1 value1 :key2 :value2 ...)
                   :period timeInSec :saveData boolean)
```

Description:

Create a network monitor with the given type, parameters, frequency and storage options.

A.12 Deleting a Network Monitor

```
(delete-experiment :sender NSAgent :receiver xxx
                   :name NetMonName)
```

Description:

Delete a network monitor named NetMonName

A.13 Terminating NSAgent

```
(terminate-agent :sender xxx :receiver NSAgent)
```

Description:

Terminate NSAgent. It will stop all network monitors and send an acknowledgement message to the sender.

```
(agent-terminated :sender NSAgent :receiver xxx )
```

Description:

Notify sender that NSAgent has terminated.

A.14 Notification about a New NetSpec Experiment

```
(new-netspec-experiments :sender NSAgent :receiver VisAgent
                         :content (NetMonName NetMonType NumHosts host1 host2 ...)
```

Description:

Notify VisAgent that a new NetSpec experiment is just started.

A.15 Notification about the Termination of an Experiment

```
(remove-netspec-experiments :sender NSAgent :receiver VisAgent
                            :content (NetMonName)
```

Description:

Notify VisAgent that a NetSpec experiment has terminated.

Appendix B

KQML Messages Implemented by VisAgent

B.1 Adding a New DPSS Client to the View

```
(new-client :sender xxx :receiver VisAgent
            :name  ClientName :pos (lat long)
            :servers (server1 server2 ...) )
```

Description:

Tell VisAgent that a new DPSS client is connected. VisAgent will update the view by adding the client node and connections to the specified servers.

B.2 Removing a DPSS Client from the View

```
(remove-client :sender xxx :receiver VisAgent
               :name ClientName)
```

Description:

Remove a DPSS client from the view.

B.3 Adding Monitoring Agents to the View

```
(registered-agents :sender router :receiver VisAgent
                   :content ( (AgentName host port status)
                              (AgentName host port status) ...)
```

Description:

Tell VisAgent about the names and status of all agents registered with the router.

B.4 Adding a New NetSpec Experiment to the View

```
(new-netspec-experiments :sender NSAgent :receiver VisAgent
                         :content (NetMonName NetMonType
                                   NumHosts host1 host2 ...) )
```

Description:

Notify VisAgent that a new NetSpec experiment is just started.

B.5 Removing an Existing NetSpec experiment from the View

```
(remove-netspec-experiments :sender NSAgent :receiver VisAgent
                            :content (NetMonName)
```

Description:

Notify VisAgent that a NetSpec experiment has terminated.

B.6 KQML Messages Sent to ServerMonitor

In addition, VisAgent also sends KQML messages to the ServerMonitor to collect information about a DPSS host and the whole DPSS systems.

B.6.1 Getting Information about a DPSS Master

```
(ask-one :sender xxx :receiver ServerMonitor
        :content (GetMasterInformation SystemName systemName))
```

Description:

Get information about the DPSS master and servers in a system.

```
(reply :sender ServerMonitor: receiver xxx
       :content (MasterInformation SystemName systemName isUp
                         NumberOfUsers numUser NumberOfDataSets numDataSet
                         TotalMemory totalMem TotalMemoryUsed totalMemUsed
                         (ServerName server1 TotalMemory serverMem
                                     TotalMemoryUsed serverMemUsed) ...
                         (UserName userName ProgramName progName
                                   IP_Address ipAddr HostName host
                                   SessionID session
                                   DataSetID dataSetID DataSetName dataSetName
                                   NumberOfServersUsed numServer
                                   NamesOfServersUsed server1 ...)...))
```

Description:

Information about the DPSS hosts in the system. VisAgent uses the information about the users/clients to keep an updated view.

B.6.2 Getting Information about a DPSS Host

(ask-one :sender xxx :receiver ServerMonitor :content (GetHostInformation System-Name systemName NodeName nodeName))

Description:

Get information about a DPSS host.

```
(reply :sender ServerMonitor :receiver xxx
       :content (HostInformation SystemName systemName NodeName nodeName
                                 HostIsUp UpTime upTime
                                 NumberOfUsers numUsers Load load
                                 Pages pages DiskActivity  d1 d2 d3 d4
                                 UserCPU userCPU SystemCPU systemCPU))
```

Description:

Information about a DPSS host. VisAgent displays this information when user clicks on a DPSS node.

Appendix C

Mapping Configuration File

The following mapping configuration file was used to configure the visual element mapping in the VisAgent (see Section 5.6.3).

```
#
#Mapping Configuration for VisAgent during MAGIC-II demonstration
#on July 14, 1998.
#For each layer, there should be 2 lines defined: first one for node,
#second one for lines.
#
#AppLayer
LABEL = DPSS;
     COLOR = LIST((MASTER, ff0000),
          (SERVER, 00ff00),
          (CLIENT, ffff00), (END));
     SIZE = FIX(3);
     SHAPE = FIX(1);
LABEL = Connection;
     COLOR = RANGE(0,4) ;
     SIZE = FIX(2);
#NetTopLayer
LABEL = Site;
     COLOR = FIX(ff00) ;
     SIZE = FIX(3);
     SHAPE = FIX(1);
LABEL = Bandwidth;
     UNIT = Mbps;
     COLOR = RANGE(0,622) ;
     SIZE = FIX(2);
#NetConnLayer
LABEL = Site;
     COLOR = FIX(ff00) ;
     SIZE = FIX(3);
     SHAPE = FIX(1);
LABEL = RTT;
     UNIT = ms;
     COLOR = RANGE(0,300) ; i
     SIZE = FIX(2);
#NetBwLayer
LABEL = Site;
     COLOR = FIX(ff00) ;
     SIZE = FIX(3);
     SHAPE = FIX(1);
LABEL = Throughput;
     UNIT = Mbps;
     COLOR = RANGE(10,155) ;
     SIZE = FIX(2);
#AgentLayer
LABEL = Agents;
     COLOR = LIST((JATLite, ff0000),
                  (NS_Test, 00ff00),
  (NS_Snmp, ffff00),
  (NS_Ping, 0000ff), (END));
     SIZE = FIX(3);
     SHAPE = FIX(1);
LABEL = Test;
     COLOR = LIST((NS_Test, 00ff00),
                  (NS_Snmp, ffff00),
  (NS_Ping, 0000ff), (END));
     SIZE = FIX(2);
```

Bibliography

[1] Andrew Adams, Jamshid Mahdavi, Matthew Mathis, and Vern Paxson and. Creating a Scalable Architecture for Internet Measurement. http://www.psc.edu/networking/papers/nimi.html.

[2] Joel Apisdorf, Kevin Thompson, and Rick Wilder. Oc3mon: Flexible, Affordable, High Performance Statistic Collection. http://www.nlanr.net/NA/Oc3mon.

[3] Brian Buchanan, Douglas Niehaus, Raghavan Menon, Sachin Sheth, Yulia Wijata, and Sean House. The Data Stream Kernel Interface. Technical Report ITTC-FY98-TR11510-04, Information and Telecommunication Technology Center, June 1998.

[4] CAIDA. Cooperative Association for Internet Data Analysis (caida). http://www.caida.org.

[5] CAIDA and NLANR. Caida Measurement Tool Taxonomy. http://www.caida.org/Tools/taxonomy.html.

[6] Robert L. Carter and Mark E. Crovella. Dynamic Server Selection using Bandwidth Probing in Wide-Area Networks. Technical Report BU-CS-96-007, Boston University, Boston, MA, March 1996.

[7] Robert L. Carter and Mark E. Crovella. Measuring Bottleneck Link Speed in Packet-Switched Networks. Technical Report BU-CS-96-006, Boston University, Boston, MA, March 1996.

[8] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). RFC 1157, 1990.

[9] University of Stanford Center for Design Research. JATLite: The Java Agent Template. http://java.stanford.edu.

[10] Cisco. Netflow Interface. ftp://ftp-eng.cisco.com/pub/NetFlow/FlowCollector_2.0/demo/NFC2_0.README, July 1998.

[11] Les Cotrell. Network Monitoring Tools. http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html. Network Monitoring Force Group of Stanford Linear Access Control.

[12] Les Cottrell and Connie Logg. Network Monitoring for the LAN and WAN. Talk given at ORNL, http://www.slac.stanford.edu/grp/scs/net/talk/ornl-96/ornl.htm, June 1996. Stanford Linear Accelarator Center Computing Services Group.

[13] Brian Tierney et al. The Netlogger Methodology for High Performance Distributed Systems Performance Analysis. http://www-itg.lbl.gov/DPSS/logging/. Future Technologies Group, Lawrence Berkeley National Laboratory.

[14] Brian Tierney et al. An Overview of the Distributed Parallel Storage System (DPSS). http://www-didc.lbl.gov/DPSS/Overview/DPSS.handout.fm.html.

[15] T. Finin et al. DRAFT Specification of the KQML Agent Communication Language. unpublished draft, 1993.

[16] Patricia Gomes Soares Florissi and Yechiam Yemini. Management of Application Quality of Service.

[17] The Cooperative Assocation for Internet Data Analysis (CAIDA). The Genmap Package. http://www.caida.org/Tools/Genmap/.

[18] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit.

[19] Anil Gopinath. Performance Measurement ans Analysis of Real-time CORBA Endsystems. Master's thesis, University of Kansas, Lawrence, Kansas, June 1998.

[20] D. Hughes. Mini SQL: A Lightweight Database Server. http://www.hughes.com.au/library/msql1/manual. Bond University, Australia.

[21] Van Jacobson. libpcap: the Packet Capture Library. ftp://ftp.ee.lbl.gov/libpcap.tar.Z.

[22] Van Jacobson. Congestion Avoidance and Control. In *Proceedings SIGCOMM'88 Symposium on Communications Architectures and Protocols*, pages 314–329, Stanford, CA, August 1988.

[23] Van Jacobson. traceroute. ftp://ftp.ee.lbl.gov/traceroute.tar.Z, 1988.

[24] Van Jacobson. pathchar: A Tool to Infer Characteristics on Internet Paths. ftp://ftp.ee.lbl.gov/pathchar/msri-talk.pdf, 1997.

[25] Rick Jones. Netperf. http://www.cup.hp.com/netperf/NetPerfPage.html.

[26] Roel Jonkman. The Design and Implementation of Netspec. Internal Documents, May 1995.

[27] Roel Jonkman. Remote Control and Information Protocol. Technical report, Telecommunication and Information Sciences Laboratory, University of Kansas, December 1995.

[28] Roel Jonkman and Joseph Evans. Netspec: Philosophy, Design and Implementation. Master's thesis, University of Kansas, Lawrence, Kansas, February 1998.

[29] Roel Jonkman, Douglas Niehaus, Joseph Evans, and Victor Frost. Netspec: A Network Performance Evaluation Tool. submitted to SIGCOMM'96, February 1996.

[30] Y.G. Leclerc and S.Q. Lau. Terravision: A Terrain Visualization System. Technical Report Technical Note 540, SRI International, Menlo Park, CA, March 1994.

[31] MAGIC-II. http://www.magic.net.

[32] G. Robert Malan and Farnam Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings ACM SIGCOMM'98*, 1998.

[33] Subrata Mazumdar and Aurel Lazar. Objective-driven Monitoring for Broadband Networks. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):391–402, June 1996.

[34] Measurement and Operation Analysis Team. Towards a Systemic Understanding of the Internet Organism: A Framework for the Creation of a Network Analysis Infrastructure. http://moat.nlanr.net/NAI/, April 1998. National Laboratory for Applied Network Research.

[35] Sun Microsystems. XDR: External Data Representation Standard. RFC 1014, 1987.

[36] Mark E. Miller. *Managing Internetworks with SNMP*. M&T Books, New York, 2nd edition, 1997.

[37] Shyam Murthy. A Software Emulation and Evaluation of Available Bit Rate Service. Master's thesis, University of Kansas, Lawrence, Kansas, August 1998.

[38] Mike Muuss and Terry Slatery. Ttcp. http://ftp.arl.mil/pub/ttcp, October 1985. Modified at Silicon Graphics in 1989.

[39] NLANR. The National Laboratory of Applied Network Research (nlanr). http://www.nlanr.net.

[40] Tobias Oetiker. Multi Router Traffic Grapher. http://www.ee.ethz.ch/ oetiker/webtools/mrtg/mrtg.html.

[41] Shyamalan Pather, Aaron Hoyt, and Douglas Niehaus. Netalyze. http://hegel.ittc.ukans.edu/projects/netalyze/. Information and Telecommunication Technology Center, University of Kansas.

[42] Vern Paxson. End-to-end routing behavior in the internet. In *Proceedings of SIGCOMM'96*, Stanford, CA, August 1996.

[43] Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware Mobile Programs. In *Proceedings of Usenix 1997 Technical Conference*, Anaheim, CA, June 1996.

[44] Cray Research. nettest. ftp://ftp.sgi.com/pub/src/nettest.

[45] S. Waldbusser. Remote Network Monitoring Management Information Base. RFC 1757, February 1995.

[46] Yulia Wijata. Netspec DSKI Daemon. http://www.ittc.ukans.edu/ ywijata/projects/nsdstrd/.

[47] Yulia Wijata. Netspec SNMP Daemon. http://www.ittc.ukans.edu/ ywijata/projects/nssnmpd/.

[48] Yulia I. Wijata. Capturing Network/Host State in MAGIC-II. Technical report, Information and Telecommunication Technology Center, University of Kansas, Lawrence, KS, November 1998.