NETWORK MONITORING USING DOORS AND MOBILE AGENTS

By

Sunil Upalekar

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF COMPUTER SCIENCE

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to dedicate this moment to my late Mother Mrs. Shalini Upalekar, who left us eight years back. She has always been, and always will be my biggest supporter, and my source of inspiration.

I would like to thank my Advisor Dr. Boleslaw Szymanski for his valuable guidance and contributions. Atlast but not least, my friends and colleagues John Alan Bivens and Vivek Rao for their help in completing this work.

CHAPTER 1
Network Monitoring

1.1 Essentials of Network Monitoring

1.1.1 Why Monitor Networks?

It's essential to provide Network Monitoring for the following purposes:

- Performance Tuning. Improve service by proactively identifying and reducing bottlenecks, tune and optimize systems, improve QOS, optimize investments by identifying under/over utilized resources, balance workloads.
- Trouble Shooting. Identify problems and start diagnosis/fixing, increase reliability/availability, allow user to accomplish work more effectively and maximize productivity.
- Planning. Understanding performance trends for network planning.
- Expectations. Set expectations for the distributed system performance i.e. from network through applications and see how well they are met.
- Security. Maintain a secure environment by monitoring network traffic.
- Accounting. Accounting for utilization of network resources.

1.1.2 Why is Network Monitoring so essential now?

This can be accounted to a number of factors such as:

- Distributed (client/server) environments have gained dominance over traditional centralized systems. Systems now critically rely on the network to function properly. Although distributed systems are very different from centralized environments, users expect the same, or better performance from the system.
- On systems like the Internet, we are seeing explosive growth. Unfortunately the technology to manage these networks is not growing as fast as the network itself.
- On distributed systems like the Internet we see a variety of operating system platforms, network equipment like switches/hubs/routers, different protocol stacks like TCP-IP/Netware/Appletalk, and finally a variety of network management applications. To efficiently manage all this, network monitoring becomes a very important issue.

1.1.3 How to Monitor a Network?

Currently the best way to monitor network traffic is by using Simple Network Management Protocol (SNMP). Some of the important parameters to be monitored using SNMP over routers, bridges, hubs are:

- Number of packets travelling through the network, kilobytes of data send/recieved.
- Number of packets dropped, discarded, buffer overflows.
- Critical servers, routers, bridges in the network in terms of traffic flow and data sensitivity.
- Changes in the network in terms of addition and/or removal of nodes.

1.2 Problems faced in Network Monitoring today

Today Switch-based networks are gaining in popularity over router-based networks because of their attractive price/performance ratio and their architectural flexibility. With the advent and dominance of Switched Networks, it has become increasingly difficult to do efficient network monitoring since for efficient monitoring we effectively need a probe on every switch/hub port.

Traditionally, the shared medium–such as an Ethernet collision segment or a Token Ring–has been the basic unit of network management. A protocol analyzer attached anywhere on a segment or ring sees and captures all the conversations taking place among all the nodes. An SNMP agent on a hub captures the traffic, error, and broadcast statistics for the entire segment. An RMON probe, some other kind of network monitor, or a handheld troubleshooting device is aware of all the significant events taking place on the shared medium. These devices provide the instrumentation–the fundamental data capturing job–that must be performed in order to manage the network.

Similar tools are required to instrument switched networks. The number of segments or rings has multiplied, perhaps to the stage where there are even more segments than end nodes once you count backbone segments. As a result, the required number of instrumenting devices multiplies also the cost of such probes is pretty high.

At the same time, the traffic on any given segment may have only a single pair of sources and destinations, making analysis of many problems difficult. Even a simple problem such as observing whether broadcasts are correctly received by the members of a VLAN and by no other nodes could mean attaching a protocol analyzer and a three-port repeater to every segment of the VLAN.

Switch vendors are compensating for the instrumentation deficiencies of switched networks within the hardware by creating dedicated monitoring ports to which protocol analyzers or some other monitors can be readily attached. Also what many switch vendors have done to enable manageability in switched networks, is to build RMON agents on each port. When RMON instru-

mentation is built into the basic hardware of the switch, it can be made to be reasonably powerful without detracting from the performance of the overall system.

However in an enviroment like the Internet, it is next to impossible to instrument the network if it becomes a predominantly switched network. Also managing and analyzing this data is a huge task in itself. What we need is a truely distributed application which can efficiently monitor and manage data on the network without posing much overhead on the network itself in terms network resources used.

CHAPTER 2
The DOORS Project

Distributed Object-Oriented Repositories (DOORS) has been created to support Network Monitoring in a truly distributed fashion, having minimal impact on the monitored network, and collect time-series data to be used for an adaptive control system for predicting congestion in a network. Most of the information included in this section about DOORS was presented in 1999 Conference on Autonomous Agents in Seattle, Washington [1].

2.1 What is DOORS?

DOORS constitutes a middleware between independent network managers and network routers. DOORS is designed to provide a secure, efficient, and fault tolerant method for the acquisition, management, manipulation, aggregation, and caching of network data and objects. This system was created to provide network resource information for any demanding application without leaving a heavy footprint on the network itself. The goal was to make DOORS scalable to the largest existing networks with hundreds of thousands of nodes that cannot be managed using traditional strictly hierarchical approaches.

At the core it uses Mobile Agents and a dynamic interface to support management and collection protocols such as LDAP, SNMP or NDS. Although it was built to handle many different types of data and protocols, to this date, it has just been used to support the collection, aggregation, and management of SNMP data and related objects.

DOORS tries to be smart in polling for data. Data requests are handled transparently considering the temporal requirements of the requester. In simple words, when a data request is made, if recent data is avialable, it is passed to the client at the end of each polling interval. If not, then the repository will create a new remote agent/configure an existing remote agent to poll for the requested data. The repositories are also designed to handle similar polling requests from multiple clients, if required. In anticipation of frequent requests from one or multiple sources, the DOORS may pro-actively collect network data in order to meet requests in an efficient manner. Scalability can be achieved by allowing requests outside the domain of a repository to be managed and forwarded to an appropriate cooperating repository.

2.2 Impact of data location

Network management places complex requirements on the physical location of the network data. Four major factors are:

- Performance. Performance, in this case, has to do with client queries and agent updates. For the clients and agents, their repository must be "nearby" in the sense of the physical layout of the network. This would imply that the repository resides on the same subnet or is at most a few hops away from routers that are assigned to it. However, it's unrealistic to expect a repository to reside on every subnet. One repository per some small set of physically close subnets should be sufficient.

- Avialability. For performance reasons, the optimal location of the repository would be the network region for which it is holding data. However, data for a network region should be available during periods when that region is unreachable. Therefore, the repository should be somewhere nearby without actually residing in the region.

- Bandwidth Usage. One of the main goals of DOORS is to provide its services with an absolute minimum impact on the network. We will show that DOORS using mobile agents saves on network bandwidth used while polling for network performance data.

- Data Integrity. DOORS network monitoring uses SNMP polling. In a conventional data poller, the snmp data is transferred using UDP packets. Using Agents to poll for snmp data on ther server side, and then transferring the collected data using reliable TCP connections over IP, DOORS ensures Data Integrity for the data collected from the snmp poll.

In a nutshell, DOORS using Distributed Repositories and the Agent System promotes a scaleable Network Monitoring Tool. It supports a Server-Push model, where the repositores "push" data to client(s) based upon a single initial request. This drastically reduces network traffic due to DOORS. The use of Agents makes DOORS a highly scaleable, distributed application.

2.3 DOORS Architecture

DOORS use several components to retrieve and process network data. A simple graphical representation of these components working together for a local data request (involving only one repository) is given in Figure 2.1. The purpose of each component will be given here, whereas the specifics, including what is actually used, will be described in the next section.

2.3.1 Client Interface

The Client interface is simply a device used to communicate with the repository. It was designed to be simple, so it could be both versatile, and easy to develop. It could be a small stand-alone application or part of a larger application which could involve visualization, network problem avoidance, or other "upstream" application components. It only needs to formulate and send a request in a form recognizable to the repository. Pictured as the "User Hosts" box in Figure 2.1, the client communicates with only one local repository. In our case, the client is the network

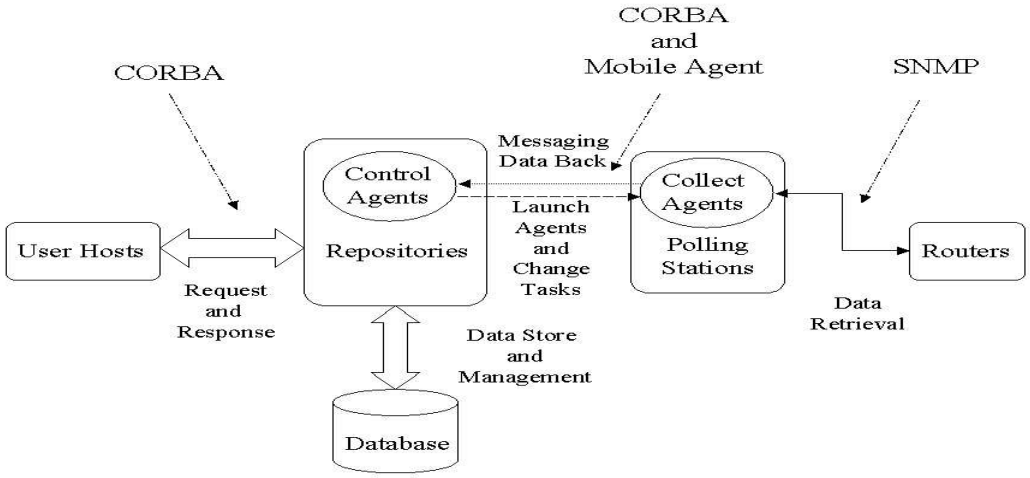

Figure 2.1: DOORS Architecture

problem avoidance application. This application will pass a request for SNMP data needed to the repository.

We have developed a Corba client using Java to communicate with the repository. It is small stand-alone application which makes requests to the repository for Router data. The details for the request made are covered in the appendix section.

2.3.2 Repository

The actual repository is responsible for the controlling of agents and coordinating requests from different clients as well as other repositories. Upon receiving a request, the repository will determine if a recent copy of the data requested exists in the storage system. If a recent copy is present, a request will not need to be issued to the router, reducing network traffic, turnaround time, and router load. Otherwise, of course, the request to the router is made.

Future work in this area involves making the repository intelligent enough to optimize system performance. If patterns are detected of certain data being requested every day at a certain time, for example, the repository can retrieve that data in anticipation of the request coming. This is an additional way in which the system decreases its response time.

2.3.3 Mobile Agents

The mobile agents are sent from the repository to a polling station to request the actual data from the destination object (typically a router).

Mobile agents give us many benefits in this implementation and offer many more benefits for future versions of this system. They help to reduce the network load by passing back only

the data necessary for the client and the repository. This is especially useful when additional processing functionality is added to the agent. In such a case, the agent will return the result of some computation or manipulation of data, utilizing the paradigm of "taking the computations to the data, rather than the data to the computations." An example of such processing is stripping off the string identification produced by SNMP and reformatting numerical data into a binary form to save the amount of bytes that must be sent back from the router to the repository.

Mobile agents also give us a great framework for updating and adding new functionality. The agent encapsulates protocols and procedures, so when a change needs to be made or functionality needs to be added, only the agent will be affected. Under any other implementation, each client and request receiving agent would have to be updated individually and consistently. Such an update becomes an expensive task to implement on larger networks.

The agents execute asynchronously and autonomously, so once an agent has been assigned to a job, the user is free to process other tasks. In addition, the agents are naturally heterogeneous. They are both computer and transport-layer independent because they depend only on their executing environment.

2.3.4 Polling Station

Polling station is a critical component, because it would be difficult to send an agent to sit on the router itself to request data. Many routers use a custom operating system and heterogeneity of platforms make the code maintenance for the data collection difficult. In addition, running the collection programs directly on the routers could introduce unacceptable load on the routers. To avoid such an overload and for security reasons, a group of routers may have a *polling station* allocated to them. Therefore, our data collecting agent travels to the polling station and from there collects SNMP data from the router with minimum network traffic. The polling station needs to run an agent daemon which can receive the agents and allow them to execute their tasks. Most agent daemons, including the Agent system that we use in DOORS, are lightweight applications, so they do not take much CPU time or memory.

2.3.5 Storage System

A sophisticated storage system is not necessary for the basic functionality of the DOORS system. We use a ObjectStore, a lightweight database system which stores objects for easy, fast retrieval. ObjectStore is to hold historical and current data retrieved with the agents, as well as meta-data used to configure parts of the system. In addition to the storage system, the repository will maintain a cache of the most recent data retrieved for quick matches with client requests.

2.4 Implementation Details

Building our prototype, we made several decisions regarding the tools used. Our primary concerns in selecting these tools were portability and extensibility.

We have chosen to use the CORBA object model as the primary vehicle for communication and management of distributed network objects that provides us with a real and usable infrastructure. Currently we are using Sun's Java version of CORBA. Java IDL adds CORBA capability to the Java-2 platform, providing standards-based interoperability and connectivity. The CORBA object model provides call back objects, which is important in the two-way communication between the client and the repository. The CORBA object model also enables us to use object written in various other languages like c/c++ if they use the CORBA interface to create their services.

2.5 Contributions

- Modified the CORBA repository code to create an SNMP Agent and send it over the network to a polling station. This was done by creating a new class in the Agent infrastructure called "AgentSender". This Agent Sender uses the "SendNRecv" class from the Agent package to send and recieve objects as files, thus facilitating the transfer of agents over the network.

- Created a proper protocol between the repository and the Agent. The development of protocol meant developing data and control messages between the "Agent" and the "AgentSender". This helped the Agent to send data and indicate an "End Of Transmission" control message.

- Created a "Data Push" protocol wherein the repository "pushes" the data recieved from the Agent directly to the CORBA client. Since the repository currently does not support any databases, this method makes the client responsible for storing data recovered from the polling station, also that no data is lost between the CORBA client and repository.

- Modified the CORBA client code, now the client requests to the repository are configured by a text file. The client reads this file to understand which router it needs to poll as well as the SNMP variables to poll for. The configuration file also includes the frequency and duration of the SNMP poll, as well as the name of the output file to be created. See Appendix C for an example of the configuration file.

- Created a proper protocol between the CORBA client and the repository. The client now, after reading the configuration file at its end (Appendix C) communicates the SNMP poll request in a single message to the repository. The repository uses this request to create an SNMP poller agent which is then send over the network to the desired polling station.

- Cleaned up both CORBA client and repository code. Improved exception handling and status messages.

CHAPTER 3
The Agent Framework

3.1 Introduction to Mobile Agents

Mobile Agents are programs, which may be dispatched from a client computer and transported to a remote server for execution. The definition of Mobile Agents is somewhat blurred, with everybody having their own ideas about an Agent System. A well-defined and thorough classification for agents has been done by Franklin and Graesser[2]

We agree upon the following characteristics that an Mobile Agent Facility should posses[3]:

- Mobile. This is the most important and required feature for any program to be deemed a mobile agent application. The agent should also display Persistence in the form of execution state necessary to support the "mobile" nature of the agent.

- Autonomous Nature. The Agent System should work without direct human interaction, and the agent should have control over its own actions and internal state.

- Adaptive/Learning. Agents can be thought of programs which encapsulate "behaviour", in addition to state. This feature allows the agent to display Adaptive/Learning nature in response to changes in its execution environment.

- Adaptive/Proactive. Agents should not simply act in response to their environment, they should be able to exhibit opportunistic, goal-directed behavior and take the initiative where appropriate.

- Communicative/Collaborative. Agents should be able to interact, when they deem appropriate, with other artificial agents and humans in order to complete their own problem solving and to help others with their activities.

Agents also tend to be small in size. They do not, by themselves, constitute a complete application. Instead, they form one by working in conjunction with an Agent Host and other agents. In many ways, agents are of the same scope as Java applets – small and of limited functionality on their own.

Agents make an interesting topic of study because they draw on and integrate so many diverse disciplines of computer science, including objects and distributed object architectures, adaptive learning systems, artificial intelligence, expert systems, genetic algorithms, distributed processing, distributed algorithms, collaborative online social environments, and security.

3.1.1 Why Mobile Agents?

Agent technology solves, or promises to solve, several problems on different fronts[4].

- Mobile agents attempt solve the client/server network bandwidth problem. In a normal scenario, web-servers handle multiple clients at the same time. The network resources like bandwidth are overloaded during each client/server request/reply. Agents enable the client application to migrate to the server side where the desired network service is avialable so that the client-server communication can be done locally over the same machine as the server, thereby saving on network bandwidth.

- The design of a traditional client/server architecture is pretty rigid. The architect, at design time, has to make decisions about where a particular piece of functionality will reside based on network bandwidth constraints, network traffic, transaction volume, number of clients and servers, and many other factors. Bad design decisions/estimates will hamper performance of the application. Unfortunately, once the system has been built, it's often difficult if not impossible to change the design and fix the problems. Architectures based on mobile agents are potentially much less susceptible to this problem. Fewer decisions must be made at design time, and the system is much more easily modified after it is built.

- To elaborate further upon the above point – software update process becomes "centralized". Any changes to the agent program can be done at a central point from where the agent can migrate across various nodes on the network. Thus the system updation becomes easy to manage and we can ensure that the changes propogate across the desired nodes across the network.

- Agent architectures also solve the problems of intermittent or unreliable network connections. In most network applications today, the network connection must be alive and healthy the entire time a transaction or query is taking place. Agents can be designed to handle unreliable networks in various ways. In case that the current connection between and Agent and it's host goes down, the agent can try to establish a new connection to resume the transfer of data.

- Agent technology also attempts to solve via adaptation, learning, and automation, the age-old problem of getting a computer to do real thinking for us. It is a difficult problem. The artificial intelligence community has been battling these issues for two decades or more. The potential payoff, however, is immense.

3.2 Agents and the Internet

The Internet is probably the most complex and unpredictable environment ever faced by Application designers. We can visualize the Internet as a massively parallel/distributed computer

system made up of millions of heterogenous computing environments which can interact with each other either directly or indirectly when required.

Today software systems developed using object-oriented languages and applications seem unable to model and tackle the Internet environment in a natural and clean way[5]. On the other hand, the Agent System paradigm is ideally suited to tackle the problems posed by the Internet.

We can look upon an Agent System as a "Society" of agents, each agent having its own locus of control of the whole system. Each agent will try to accomplish its task(s) by interacting with its environment, and with other agents in order to access information or services that it does not possess, or to co-ordinate its activities within the Agent System. Agent Systems are truely distributed, robust, automomous, adaptive as well as pro-active, all these qualities of an Agent System offer a computational model ideally suited for the Internet environment.

Agents can act as representatives of the user over the Internet. As an individual, the agent is a software system with its own technology/structure/purpose. However on the outside, it is a member of a "society", interacting with other individuals, acessing remote resources, and exploiting the social infrastructure. This is what makes the Agent System a radical departure from the more traditional software systems[6]

3.3 The Agent Architecture

A Mobile Agent System is a framework to facilitate the deployment of mobile agents on the network. It provides the functionality of mobile code.

In order to achieve this objective the system was designed to consist of 4 major components.

1. The underlying communications layer which provides a stable platform for communication between different components of the system.

2. A Server daemon running on nodes all over the network. This server daemon is required for a process to migrate to a particular node. The daemon recreates an agent and provides a favorable environment for it to execute.

3. A discovery and lookup protocol is necessary to dynamically lookup all the Servers on the network. This is provided by a lookup package with a centralized server called SuperServer.

4. Various support packages are provided to develop and deploy a Mobile Agent. This makes the process of developing an agent and keeping control of it very simple.

3.3.1 Basic Working

All these components work together so that an agent created at an "Agent Sender" looks up for servers registered at the "SuperServer" and migrates to it. This is a mobile agent and now continues execution on the server – where its remoteProcess function is automatically called. The agent can carry out its operation and then migrate to another server. It can also communicate

to its Agent Sender - sending it some intermediate information. The Agent Sender can also track down the agent in the network and communicate with it. When the agent has finished its execution at all hosts it can return to its Agent Sender for post processing.

3.3.2 Features

3.3.2.1 Efficiency

A mobile agent system is much more efficient than a client-server system when highly complicated operations are involved. Using a mobile agent system minimizes the network traffic required.

The framework maximized the efficiency of the system by fine tuning the process of agent creation. Class files for agents are stored in a database based on their unique UID. If the agent is of an unknown class it is downloaded again and stored. But if the class has been downloaded before it is loaded from the database.

3.3.2.2 Fault Tolerance

This system is highly distributed and forces us to implement highly robust fault tolerance. Working in a network a Server may crash causing the agent to not be able to propagate further, or the server at which the agent currently is - may itself crash destroying the agent. These problems are handled gracefully and circumvented where possible.

3.3.2.3 Security

Security becomes an important issue in the system because an agent has downloaded code and if it is malicious or badly written - it can cause harm to the server. This problem is solved by implementing the Sand-Box model. So an agent can only execute operations which it is allowed to by the policy of the server.

3.3.3 Detailed description of components

1. lookup: The lookup package contains classes for server registration and discovery. Using these classes it is not necessary for the servers to know about each other in advance. The superserver is always running as a pre-specified service. Whenever a server starts up it registers at the Superserver. Having registered it can be accessed by any other Agent / Server by looking up at the SuperServer.

   The SuperServer also provides the capacity to filter requests based on some specified parameters, to lookup servers that match some criteria - instead of looking up based on the server name. Hence servers can be dynamically found.

   This concept of “Service Lookup” is an inspiration from the Java “Jini” technology. We this that this service is ideally suited for an Agent Architecture, since it offers a lot of flexibilty to use existing services dynamically on the web, as well as offer new services on the fly.

The Agent Lookup service is probably our biggest contribution in making this framework a truely "Generic" package. Thus any new services to be added can be easily incorporated into our framework by registering themselves with the SuperServer.

2. IO: The package IO provides synchronous object based communication between components. The most important class in the package is "SendNRec". This class provides a layer for transfer of Messages, Objects, integers etc. between components. The address of a component is specifed in a data structure called AgentAddress.

   Again Java is ideally suited for building such an I/O component because Java supports Object Serialization. This feature enables us to transfer object between hosts and helps preserve "Soft State" of the program.

3. Server: The Server is always running on all the nodes. It registers at the SuperServer for it to be accessible. It then loads a list of all the already existing class files so that they need not be downloaded again. It then listens for requests from agents or agent senders.

   These servers are our "Service Providers". For example, a network monitoring application can "lookup" a server which offers SNMP services and then send an Agent to that server to perform snmp polling for the application.

   The server creates a new AgentThread for each request from a migrating agent. This thread handles the downloading of the agent and its recreation. Then it calls the agent's "remoteProcess()". The actions specified in this function are carried out. After the completion of this process - the agent calls the agent's "getNextHost()" function to find out where the agent wants to go next. Given a valid host it forwards the agent to that host. Otherwise it calls the agent's "getHome()" to find where it came from - and sends it back to its agent sender.

   The agent thread is responsible for the Sandbox protection given to the server. Thus an application can have a "Many to Many" relationship for the agents that it creates versus the server avialable to "host" the agent. This provides a true distributed nature to the application. What this does, is harness the power of the Internet as a highly distributed computational machine effeciently.

4. Agent: The Agent package provides some classes necessary to build an agent. Any class trying to be an agent has to implement the RemoteExecution interface. It is expected to implement some functions like "setHostList()", "remoteProcess()", "getNextHost()", "getHome()", "processMessage(')" etc. A Service Adapter called "Remote Execution Adapter" is provided so that an agent can be created by extending this adapter.

   Again this is a huge factor in the "Genericity" of our framework. Any new application which wants to "upgrade" itself as an Agent Application has to extend the remote execution adapter class. The Agent gramework will take care to the remaining details.

Agent Sender is a class that provides the functionality for deploying an agent. It creates and agent - initializes it and sends it to its first destination. The Agent Sender then listens on a port waiting for any communication from the Agent. The Client program in the meanwhile can perform other actions. This asynchronous style of operation gives a lot of power to the agent system.

The Agent Sender can also initiate communication with the agent. It can track the agent which may have gone to another server. Commands can be sent which can dynamically decide the path of the agent.

The agent system here - provides a new and powerful tool for distributed object systems. The concept may take on a significant importance in the future.

Detailed documentation for the code is present in the form of "Java Docs", and can be downloaded along with the Agent Package.

3.4 Contributions

- Improved the concept of service lookup via the "SuperServer" in the Agent package. Each Server or polling station is now responsible for specific routers in the network. When a polling station is started, it's registration process with the "SuperServer" was modified to include a list of routers for which the polling station is responsible. Thus now when an "AgentSender" has to poll a specific router, it can know which polling station to send the poller agent via the "SuperServer"

- Modified the Agent Server i.e. polling station's code so that now the polling station can be configured to understand which routers it is reponsible for, as well as on which host is the "SuperServer" is running, so that the Server can register itself to the "SuperServer". A sample of the Server configuration file is included as Appendix D.

- Improved the registration process between the "SuperServer" and the "Server". The registration process was customized for SNMP polling. The "SuperServer" code was modified so that it maintains a lookup between Routers and Polling Stations.

- Wrote and tested the code for SNMP Agent. This Agent is the data collector for the DOORS project. This agent can be configured easily to poll for one or more routers and for desired SNMP variables. The polling times and intervals are also configurable.

- Wrote and tested the code for "SnmpPoll.java". This class is used by the Snmp Agent to poll for SNMP data. It uses the AdventNet SNMP package.

- The download of class files between the Agent Sender and the Agent Host is a big overhead for the DOORS application in terms of bandwidth used. This download process was optimized by creating a hashed "classes.dat" file on the Server side which maintains a list of Agent

Classes as per their name and data of last modification. The Server first check this file to see if it already has the class files or not before requesting the Agent Sender for the same. This means, once the Agent Class file is downloaded, it will not be downloaded again for every new request, unless the class file changes at the Agent Sender side.

- Optimized the bandwidth used for communication between the Agent and the Agent Sender. Initially the communication included "sync" messages between the Agent and the Agent Sender after each data transfer. After proper testing and analysis, it was seen that these "sync" messages are infact unnecessary and use up valuable bandwidth. The removal of these control messages reduced the bandwidth usage by the DOORS application.

- Cleaned up Agent Package code. Improved exception handling and status messages.

CHAPTER 4
Network Monitoring for the Doors Project

4.1 Testing

As described in the earlier chapter, we retrieve SNMP data from workstations for the routers that are targeted in client queries. SNMP could be used to retrieve data from a distance instead of a more complex system such as the one that we have described in this paper. However DOORS can provide fault tolerance, ease of data management, and ability to aggregate data and requests.

To compare the advantages of using Agents over normal polling methods, tests were run collecting SNMP data directly from a router that was one hop away from a client designed using AdventNet Java SNMP package. The same tests were run using a DOORS prototype that included Java ORB, Agents, and the SNMP package. The test were run on an isolated network involving our lab machines and routers is given in Figure 4.1.

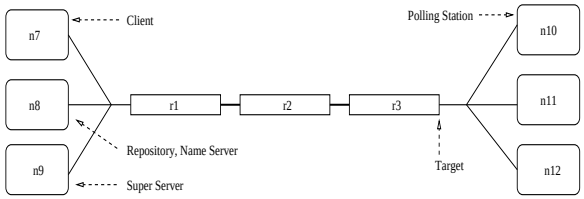

Figure 4.1: Network Topology

Our testbed consisted of six machines running FreeBSD and three Cisco 2500 series routers in a private network. The topology can be seen in Figure 4.1.We configured two subnets of three machines connected via three routers which in turn are connected to each other through high speed serial links and running the RIP routing protocol. We always collect the snmp data from the Ethernet interface of router3.

For our tests to measure the traffic incurred across these networks, we monitor the bandwidth seen (IfInOctets) by the serial1 interface of router2 (r2 in the middle). For DOORS experiments, the snmp polling station namely the host server was running on machine "n1.ns.rpi.edu". The Corba repository and the nameserver were on "n5.ns.rpi.edu". The SuperServer on machine "n6.ns.rpi.edu" and finally the client or the snmp polling application on "n4.ns.rpi.edu".

We wanted to compare network bandwidth utilized by normal snmp polling methods over DOORS. To this end, we created a snmp poller client using the Adventnet snmp package to request and receive data. For experiments using the snmp poller, the SNMP poller client runs on

Table 4.1: IfInOctets/Second with normal snmp Poller and DOORS

Polling Interval (Seconds)	DOORS (IfInOctets/Second)	SNMP Poller (IfInOctets/Second)
3	0.066031818	0.074889182
5	0.053046956	0.058714725
7	0.047435567	0.051179838
10	0.043161447	0.045875865

"n7.ns.cs.rpi.edu" polling the same router "r2".

We used a snmp bandwidth probe created by us to monitor the ifinOctets on the serial interface s1 of router r1 using the topology given in figure 4.1. Table 4.1 shows bandwidth usage of DOORS versus a normal snmp poller for various polling intervals. We calculated the change in ifinOctets in the given time to determine ifinOctets per Second using both tests. This was effectively a bandwidth calculation for the network with both methods.

Bandwidth comparisons were made between the normal snmp method and the same requests obtained with DOORS (Figure 4.5). Tests were run for three minute, five minute, seven minute and ten minute intervals requesting the following SNMP variables from the r3 router: {2.2.1.10.2 (*ifInOctets*), 2.2.1.16.2 (*ifOutOctets*), 4.3.0 (*ipInReceives*), 4.9.0 (*ipInDelivers*), 4.10.0 (*ipOutRequests*)} For more information about SNMP variables, please see RFC 1213 [7].

4.2 Results

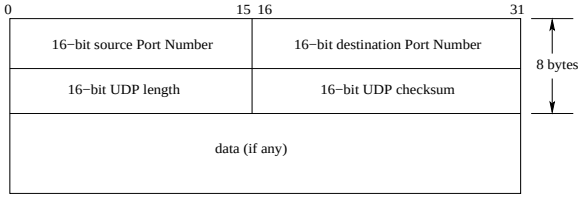

Figure 4.2: UDP header

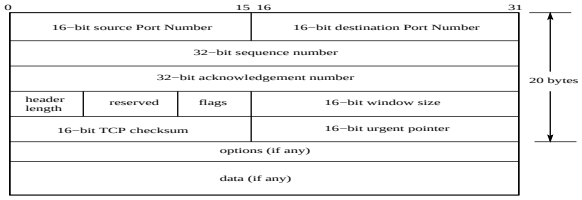

Figure 4.3: TCP header

The DOORS system uses TCP for agent to repository communication while the normal SNMP method uses UDP. Our agent maintains the same connection with the repository, so the

connection handshakes can be amortized over the life of the connection. Therefore, the bandwidth difference can be related to differences between TCP and UDP header data. As shown in figures 4.2 and 4.3, UDP headers are typically 8 bytes while the TCP headers are usually 20 bytes. Therefore, the relationship between the TCP data packet and the UDP data packet is the following:

$$Psize_{tcp} = 20 + C \tag{4.1}$$

$$Psize_{udp} = 8 + C \tag{4.2}$$

In these formulas C is the size of the data payload (which should be the same). An addition difference that should be mentioned here is the SNMP headers that will be sent using regular SNMP packets, compared to the overhead of our agent data packets. As seen in figure 4.4, The SNMP header are approximately 30 bytes, while the overhead from our agent data messages is 4 bytes. Therefore, the relationship between the typical DOORS data packet and the typical SNMP polling packet is shown in the following approximations. Thes e equations are approximations, because we are currently investigating the internal value representations used in SNMP packets which we think may be different that the strings we send from our agent.

$$Psize_{doors} \approx Psize_{tcp} + 4$$

$$Psize_{snmp} \approx Psize_{udp} + 30$$

Given that the total bandwidth used can be given by multiplying the message size by the number of messages sent, we can obtain the following formulas:

$$Band_{doors} \approx Psize_{doors} * Msgs_{doors} \tag{4.3}$$

$$Band_{snmp} \approx Psize_{snmp} * Msgs_{snmp} \tag{4.4}$$

Because the doors system sends an agent across only once for the life of the request, we assume its cost is amortized and we do not consider it in our calculation. However, once the agent is there, it will send all of the results without the client sending a separate request message each time. Due to this, DOORS uses about one-half of the messages the normal SNMP method uses. This could be described in the following formulas:

$$Msgs_{doors} \approx \frac{Msgs_{snmp}}{2} \tag{4.5}$$

We can now substitute for formulas 4.3 and 4.4, to see the following:

$$Band_{doors} \approx \frac{(20 + C + 4) * Msgs_{snmp}}{2}$$
$$Band_{doors} \approx \frac{(24 + C) * Msgs_{snmp}}{2}$$

At this stage we can look at the approximate bandwidth usage value from DOORS and that from normal SNMP and see that DOORS' usage value is considerably less:

$$\frac{(24 + C) * Msgs_{snmp}}{2} < (38 + C) * Msgs_{snmp} \tag{4.6}$$

leading us to believe,

$$Band_{doors} < Band_{snmp} \tag{4.7}$$

As we can see by the graph in Figure 4.5 and our conclusions in Equations (4.6 and 4.7), DOORS can be an effective solution to lighten the footprint of any network management application. If the number of clients grow, the relative bandwidth used by DOORS will remain almost constant while the normal SNMP clients will continue increasing bandwidth usage.

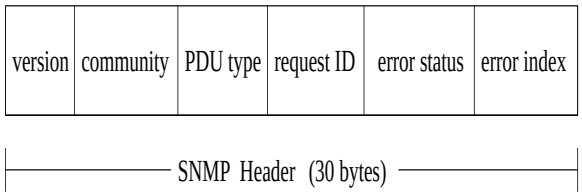

Figure 4.4: SNMP header

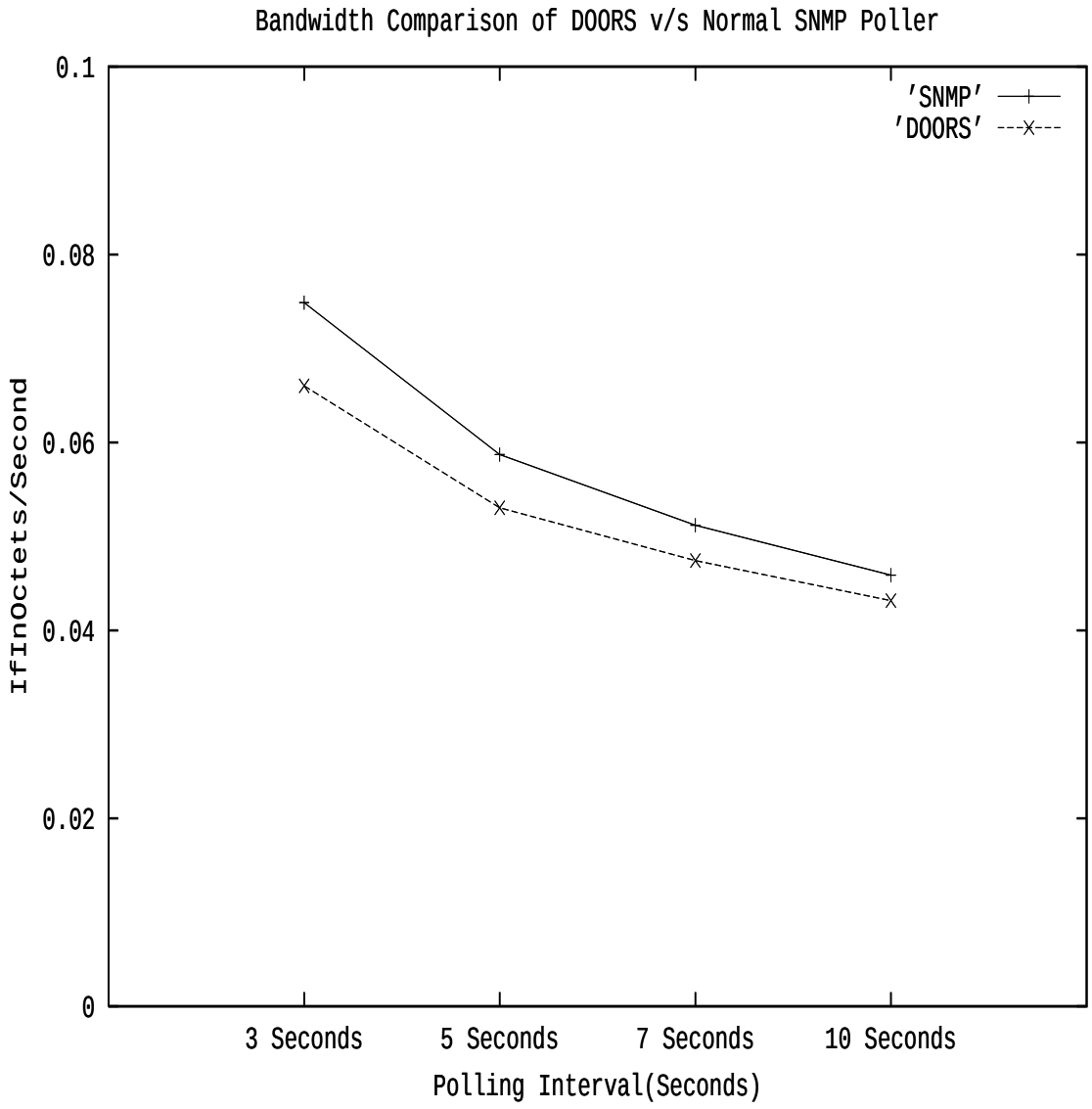

Figure 4.5: Comparison Chart

BIBLIOGRAPHY

[1] J. Alan Bivens and Li Gao and Mark Hulber and Boleslaw Szymanski. "Agent-based Network Monitoring". Proceedings from the Agent based High Performance Computing Symposium sponsored by ACM SigART, Seattle, Washington, May 1999.

[2] Stan Franklin and Art Graesser. "Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents". Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[3] N. R. Jennings and M. J. Wooldridge. "Applications of Intelligent Agents". Agent Technology: Foundations, Applications, and Markets: (1998)

[4] Todd Sunstead. "An Introduction to Agents": An Article in JavaWorld, July 1, 1998.

[5] G. Booch. "Object Oriented Analysis and Design(Second Edition)". Addison Wesley, Reading(MA), 1994.

[6] Franco Zambonelli, Nicholas R. Jennings, Andrea Omicini, Michael Woodridge. "Agent-oriented Software Engineering for Internet Applications", July 04, 2000.

[7] K. McCloghrie and M. Rose, "Management Information Base for Network Management of TCP/IP-based internets" RFC 1213, March 1991.

APPENDIX A
Installing DOORS

- The source code for the DOORS software is located under "/cs/upales/doors.tar.gz". Copy it to your home directory and install the same using:

  ```
  zcat doors.tar.gx | tar xvf -
  ```

- This will create a doors directory under your home directory. We have an existing set of binaries for Solaris and FreeBSD build under the doors directory as "./doors/OSNAME/bin".

- The source code files are located under "./doors/source". Compilation instructions can be found in a file called compile.instr.

- In your home directory, create a "classdir" to hold the agent classes. Make sure your classdir is diff than the directory in which your actual agent class files are sitting.

- Add to your classpath the following: " $HOME/doors$/OSNAME/bin/"

- To run the super server, run it under your home directory – the one in which you created the "classdir". Type:

  ```
  java msu.magent.lookup.SuperServer
  ```

- To run the agent server, run it under your home directory, using:

  ```
  java msu.magent.LocalServer.Server <portnumber> <config-file>
  ```

  There is sample config file under my directory: upales/pollconfig. The first line is the machine on which the superserver is working, and the next lines are the routers which need to be polled.

- Now go to, upales/thesis/doors/bin, or your local directory where you have all the class files and type:

  ```
  ./tnameserv -ORBInitialPort 1050 > namingfile &
  ```

- Run from the same directory:

  ```
  java ProNet_serverCB -ORBInitialPort 1050 -h <superserver host name>
  ```

- Run:

  ```
  java ProNet_client -n config1 namingfile
  ```

  config1 contains sample configuration for polling.

APPENDIX B
Sample Source Code for writing Agent Applications

Shown below is an example of converting your Java application into an Agent using the "Generic Agent Framework".

```
// Author: Sunil Upalekar
import java.io.*;
import java.util.*;
// MSU Agent package
import msu.magent.Agent.*;
import msu.magent.LocalServer.*;
import msu.magent.lookup.*;
import msu.magent.io.*;

//
// Class: SnmpAgent extends RemoteExecutionAdapter implements Serializable
// Function: Enables mobility using above mentioned classes, and creates a
// poller process depending upon the configuration specified
//

public class SnmpAgent extends RemoteExecutionAdapter implements Serializable {

    // SNMP Data Poller
    SnmpPoll poller;

    //pipelines via which Agent gets data from poller
    private PipedOutputStream out;
    private PipedInputStream in;

    // Agent Input Stream to read the Poller ouput
    DataInputStream inStream;

    // A count for the Agents instantiated
    private static int agentCnt = 0;

    // Configuration variables
```

```
    public String requestID;
    public String hostRouter;
    public String[] oIDs;
    public int pollInterval = 0;
    public int pollTime = 0;
    public int count = 0;

    // Agent Constructor
    SnmpAgent(Vector args) {

// Update the Agent Count
agentCnt++;

// Set Configuration as per arguments passed
parseArgs(args);
System.out.print("Agent Constructor called with Arguments:");
for(int i = 0; i < args.size(); i++)
    System.out.print(" "+args.elementAt(i));
System.out.println();
    }

    void parseArgs(Vector args) {

//
// I am assuming a very strict order of arguments to be passed to the Agent
// This is how the Agent should be called:
// SnmpAgent abc = new SnmpAgent(requestID, hostRouter, pollinterval, count, oIDs)
//
//
requestID = (String)args.elementAt(1);
hostRouter = (String)args.elementAt(3);
pollInterval = Integer.parseInt((String)args.elementAt(5));
pollTime = Integer.parseInt((String)args.elementAt(7));
count = pollTime/pollInterval;

oIDs = new String[args.size()-9];

for(int i=9; i < args.size(); i++) {
```

```
    // Parse out all the oID strings
    oIDs[i-9] = (String)args.elementAt(i);
}
    }

    // For Mobile Code, we need to override this method from the RemoteExecution Adapter
    public void remoteProcess() {

// create a byte Array to recieve the poller data
byte[] bArray = null;
int len = 0;

// A string to hold the message from the poller
String msg = null;

try {
    // Create a new SendNRec instance to send/recieve remote data
    SendNRec snr = new SendNRec();

    // Increment the Agent Hop Count
    hop++;

    // Set up the pipes and streams
    out = new PipedOutputStream();
    in = new PipedInputStream(out);

    inStream = new DataInputStream(in);

    // Create a snmp Poller with the required settings
    poller = new SnmpPoll(requestID, hostRouter, oIDs, pollInterval, count, out);

    // Start the poller to send data to the Agent
    System.out.println("Agent: Initiated Polling!");
    poller.startPolling();

    // Find out the Poller Status
    if(poller.getPollingStatus()) {
```

```
System.out.println("Agent: Polling active...");
System.out.println("Agent: Polling Interval: "+poller.getPollInterval());
    }

    int count = 0;
    // Keep reading Data from the poller till required
    while(true) {

// Find out the Message Length
len = inStream.readInt();

// Check if there is no message
if(len == 0) break;

bArray = new byte[len];

// Read the message
inStream.readFully(bArray, 0, len);
msg = new String(bArray, 0, len);

// Print out the message recieved
System.out.println("SnmpAgent Msg: len : " + len + " Msg: " + msg);

// If it's not the final message
try {
    if(msg.compareTo(SnmpPoll.FIN) != 0) {

// Send the message to the process which created the Agent
System.out.println("Agent Home: "+home);

snr.connect(home);
snr.sendMsg("msg");
snr.sendMsg(new String(msg));
snr.disconnect();
    }
    // Else convey end of transmission
    else {
snr.connect(home);
```

```
snr.sendMsg("exit");
//snr.disconnect();
    }
} catch (SyncException se) {
    System.out.println("IO SyncException:"+se.getMessage());
    se.printStackTrace();
}
    }
} catch(IOException e) {
    System.out.println("IOException:" + e.getMessage());
    e.printStackTrace();
}
    }

    // Use the below code for testing purposes
    /*
    public static void main(String[] args) {
Vector v = new Vector();
v.add("-I");
v.add("1234");
v.add("-R");
v.add("cobol.cs.rpi.edu");
v.add("-i");
v.add("2");
v.add("-T");
v.add("8");
v.add("-O");
v.add("2.2.1.10.2");
v.add("2.2.1.16.2");
v.add("4.3.0");
v.add("4.9.0");
v.add("4.10.0");
SnmpAgent agent = new SnmpAgent(v);
agent.remoteProcess();
    }
    */
}
```

APPENDIX C
Sample Configuration File for DOORS Client

```
SEGMENT
10.1.7.1 // Router to be polled
router.out //  Output file for data collected
5 // Polling interval in Seconds
600 // Number of Polls
//SNMP Variables to be polled follow...
2.2.1.10.2
2.2.1.16.2
4.3.0
4.9.0
4.10.0
END
```

APPENDIX D
Sample Configuration for Agent Server

```
10.1.1.12 // SuperServer IP address
10.1.3.1 // Router IP Address (You can add Multiple router entries!
```