Monitoring & Management Service over Distributed Environments

By
Afrasyab Bashir

Project report for partial fulfillment of the requirements of MCS/NUST
for the
award of the B.E degree in Software Engineering

Department of Computer Science
Military College of Signals
Rawalpindi

May 2003

Acknowledgement

To determination that wins the fortune and of course to my parents and everyone who helped and who didn’t.

Abstract

Monitoring resources is an important aspect of the overall efficient usage and control of any distributed system. The resources of interest can include all manner of networked devices, from a remote sensor or satellite feed through to a computational node or a communications link. This generic, open-source resource monitoring architecture has been specifically designed for the Grid being an ultimate implementation yet defined for distributed architecture. A wide-area distributed system such as a Grid requires that a broad range of data be monitored and collected for a variety of tasks such as fault detection and performance monitoring, analysis, prediction and tuning. A range of tools has been developed for monitoring distributed resources.

The system is based on the basics of Services Oriented Architecture, Java technologies (jini, applets, servlets and JDBC) a SQL database and SNMP (agents and objects). Unlike many other monitoring systems, it is designed to monitor both Grid resources and processes, rather than only the applications that execute on a Grid. It is capable of registering interest in events, remotely observing devices, as well as gathering and displaying monitoring data.

Contents

List of Figures

Chapter One

1.1 Introduction

Primary aim of networking, may it be a simple network of two computers or a distributed network encompassing the whole globe i.e. Internet, is to share resources. The resources range from hardware computing & telecommunication components to software application & services. As far as technology goes, the Internet is the state of the art for many people in the world. It has facilitated innovations that make possible new services empower employees, consumers and citizens with access to the latest information, allowing organization to cement new forms of strategic partnerships and enables new forms of sharing.

World Wide Web [1] has spawned entire new industries and ways of work. It has led to the creation of a brand new medium for collaboration, interactions and sharing, but the main resource to be shared remained files in shape of web-pages and emails. And this limitation of sharing the resources became a background for the next evolutionary step to raise the bar for the positive benefits of communication; this step increasingly appears to be related to Grid computing. Grid computing makes possible a phenomenon that is beyond enhanced collaborative communications or the sharing of information, it allows for communities to share actual computing resources as they tackle common goals. These communities can link their data, computers, sensors and other resources into a single virtual environment. Every resource can be wrapped into a service that is accessible across the hard boundaries of geography and the soft borders of institutions.

Ultimately, the Grid [2] will open up storage and processing powers the same way that the Web has opened up access to content. Computing will become a utility just as any other utility, and will be ubiquitously accessible. This can break the desktop prison and individual users could use grid enabled hand-held devices, mobile phones, public access points or other as yet undiscovered mechanisms to access computing resources and services that could exist anywhere in the world.

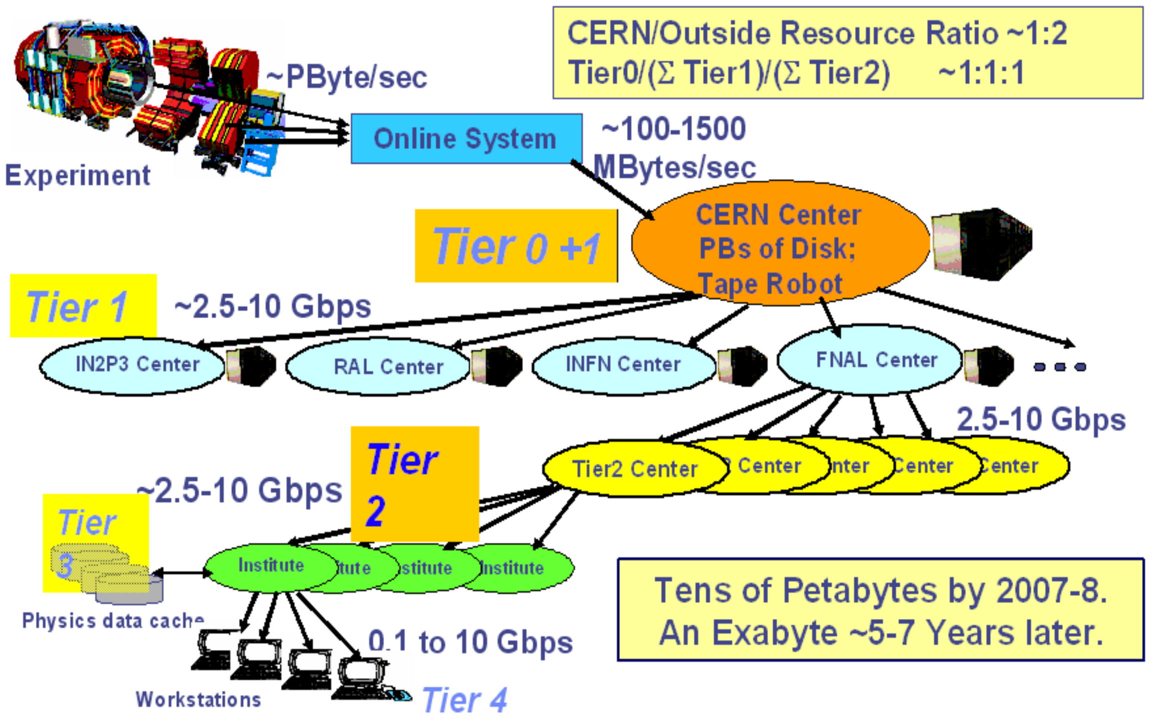


Figure 1-1 Hierarchical Structure of LHC Data Grid

1.2 Grids: An Ultimate implementation of Distributed Systems

The word “Grid” itself was chosen as an analogy to the power grids that deliver ubiquitous access to electricity. Ultimately, there is a plan to develop an infrastructure that can provide similar access to computational power and resources as we now have to electricity, gas, and water.

The Grid [2] is an active research area that is rapidly taking shape through the efforts of leading scientists across the world. It will provide access to computing resources, databases, hard drive storage, sensors, input devices, people and information stores in a pervasive, dependant, consistent and inexpensive manner. This can have a dramatically transforming effect on the range of applications that are possible; a sensational impact on human capabilities and society is expected.

As of date, the Grid [2] as envisaged does not exist as yet, but it is on its way. The building blocks of the Grid [2] are readily available in the form of networking technologies, physical hardware, access and security policies, algorithms and techniques for applications as varied as nuclear engineering to electronic commerce. The Grid [2] will ultimately take shape as the various network communication, security and resource brokering challenges are surmounted. We will one day be able to access resources across autonomous and independent organizations that are frequently geographically distant from each other. This will be accomplished while honoring flexible, rigorous standards of remote performance and availability service levels while ensuring that local control of resources are not compromised.

1.3 Background

The typical Grid system will be built by groups that are globally or nationally dispersed but who have a pressing need to interact with each other and with remote resources. Current such a huge system development is taking place at CERN, Switzerland, who for the CMS [3] operation plus the physicists around the world need to implement GRID over Internet, thereby bringing forth the best example yet defined for distributed environments. This would be an example of data grid [4], which by its name is self evidentiary that it would be used mainly for the data storage and sharing purposes.

These Grids [4] are implemented because of different services implemented over the network and acting under certain standards. All such services [5] need the different resources for their operations to take place and these resources range from the hardware under utilization to other services loosely coupled. These services [5] can better co-ordinate amongst their own instances and with other services loosely coupled to them, only when they use historical and current status information of the resources under utilization as the knowledge base to reach a decision to act upon. This decision would provide for the best action, for enhancement of quality of service, to be taken in the situations services are acting.

Grid [2] is considered to be a single super virtual computer encompassing the whole Globe while providing different unlimited type of services. This ensemble, of heterogeneous software and hardware from multiple vendors, needs to provide different services implemented over it, the information required for their operation to run smoothly. Moreover Grids are also exhibit limitations like limited bandwidth, jitter, data misses etc. It therefore calls for a continuous monitoring of resources being shared over Grid, to collect necessary information required by different services for better efficiency.

1.4 Scope

Co-ordination amongst Higher Level Services is only possible with the monitoring of the Grid resources involved in their operation. This Monitoring Service, will provide all the information required to any of the client service over the Grid, by coupling loosely to its clients. This would ultimately provide for a decision making knowledge base, in a services oriented architecture [6], to the clients to make different decision as per the prevailing situations.

1.5 Aim

To develop a monitoring and management service for the wide area distributed system that should be able to provide the client services reliable, historical and upto date status information regarding the resources under utilization, to be used as a knowledge base for appropriate decision makings under the prevailing situations and circumstances.

References

[1] http://www.w3.org/, World Wide Web Consortium

[2] http://www.ggf.org/, Global Grid Forum

[3] http://www.cern.ch, Center for European Nuclear Research

[4] “The anatomy of the Grid”, By Ian Foster Et. Al

[5] Open Grid Service Architecture (OGSA) Specification

[6] Service Oriented Architecture Specification, http://www.jini.org

Chapter Two

Requirement Analysis

2.1 Introduction

This project is being developed as a service which has to serve both historical and real time ‘resource status information’, to its various and heterogeneous clients over the GRID, to provide the base for their efficient working, in co-ordination with the overall system. Imperative to note is that the service is being developed for distributed environment and therefore retrieval of status information would not be the only thing but its safe and reliable availability to its distant clients over dissimilar environment, is of most important consideration too.

2.2 Overview

The system is aimed to be serving different clients over distributed platform as a Network Management System (NMS). Although unlike NMSs it has to co-ordinate with its different instances over different Networks in a specified hierarchical structure. Clients that would be requiring the status information of resources may be job schedulers, data transferring and negotiating services.

An NMS is designed to serve a single network and that too of a limited number of resources due to its centralized approach, whereas this service has been planned with an approach that it could have limitless addition of resources, all the times and anytime in the distributed environment, without effecting its efficiency. This NMS would on request from its different clients would provide historical and real time status information regarding the resources under its managed scope. Different type of information regarding the resources under utilization, required by different services could be

- CPU Load
- Disk I/O
- Storage space

- Network Interface I/O
- Buffer Misses
- Bandwidth Utilization
- etc

Because of the difference between a normal network monitoring service and a service for the distributed networks, the scope of this project is not limited to the monitoring of the managed devices but to the safe and secure provision of this information to its distant and various clients at a time with the similar efficiency and speed as that of a local monitoring service.

Over distributed environment one has to look for the issues like choked bandwidth, jitter, buffer misses and limitations of different programming approaches. These limitations further add to the grave nature of scalability issues over GRID, a better distributed architecture yet defined. To cater for all this, technology and protocols that are of interests have to be used.

While keeping aforementioned in view, we find that the services oriented architecture best defines the architecture we would like to adopt for communicating and providing services to our clients. Whereas SNMP is the protocol well suited for monitoring of resources due to its implementation in nearly each and every manageable device connected to Internet.

2.3 Project Motivation

Implementation of GRID, over Internet is not possible without the monitoring of resources and therefore it makes this provides for a great motivation to go for it. Moreover, before taking up this project a detailed view of other monitoring applications was also taken, in order to find out that what all related work has already been done in this regard.

2.4 Test Bed

The monitoring system developed shall be used primarily for the LHC Data GRID[1] and secondarily it will be available for all GRID environments thus implemented anywhere. It will be a vital component for development of the LHC Computing Model [2], as well as the development of realistic "optimal" strategies for resource management and user interactions with LHC and other Grids which are both resource constrained and truly global in scope.

This system aims to provide a distributed monitoring service system using JINI/JAVA [3] and WSDL/SOAP [4] technologies. It has to act as a dynamic service system and provide the functionality to be discovered and used by any other services or clients that require such information.

The goal is to provide the monitoring information from large and distributed systems to a set of loosely coupled   "higher level services"   in a flexible, self describing way. This is part of a loosely coupled service architectural model to perform effective resource utilization in large, heterogeneous distributed centers.

The framework can integrate existing monitoring tools and procedures to collect parameters describing computational nodes, applications and network performance. It has build in SNMP support.

2.4 Functionalities Requirements

A mechanism is required to dynamically all the instances of the services running over the familiar GRID Hierarchy. It also implies that any particular instance running thousands of miles away used by a community can be discovered.

System should also be able to remotely notify the events of interest being generated in that community for a particular leased period of time, for changes in the any system. It implies that for every registered unit there should be lease mechanism i.e. the mechanism to register itself to the lookup directory service and to register the clients for a leased period of time.

Instances should be secured and able to dynamically add new instances at lower level of hierarchy thereby making their configuration and the collection of required parameters, of any interest to clients, easy.

Even a single instance to any level of hierarchy should be easily available/located by the clients and an access to lowest level i.e. nodes be provided, to authenticated clients.

It should be able to provide events of interest to registered client listeners for a leased period of time, plus parameters like connectivity value and traffic information be provided.

Additional modules that could be loaded on the request of clients may include selected real time status and historical data. History may be maintained in an RDBMS for persistency.

Active Filters to process the data and provide dedicated/ customized information to other services and clients.

Dynamic proxies to provide for the flexibility of access to the clients and compatibility with Globus Toolkit. This also means a configurable GUI to present from real-time global views of multiple farms to the evolution in time of any single parameter.

References

[1] http://www.cern.ch, The LHC Data Grid
[2] http://www.cern.ch, The LHC Computing Model
[3] http://www.jini.org, Jini Architecture Specification,

[3] http://www.webservices.org, UDDI/WSDL Specifications

Chapter Three

Literature Review

3.1 Introduction

Literature review is an important aspect to be considered during the project development. This not only includes the review of the different technologies best suitable for the project but the previous work that has already been done in this regard by different universities and organization in order to create a difference and to make it something better than the previous. Reviews of related work and technologies both are given in the following sections

3.2 Related Work

It was important to get a detailed view of all work, that has already been initiated or done by different other universities, in order to avoid the redundancy and duplicity of efforts plus it would make the work unique. To give a better view of this project, it would be imperative to provide brief information of all such related work by other universities and organizations and thereby a better picture, defining the requirements, will come forth automatically.

3.2.1 The Globus HeartBeat Monitor

The Globus Heartbeat Monitor (HBM) [1] was designed to provide a simple but reliable mechanism for detecting and reporting the failure (and state changes) of Globus system processes and application processes. The HBM module has been deprecated in the Globus toolkit.

A daemon ran on each host gathering local process status information. A client was required to register each process that needed monitoring. Periodically, the daemon would review the status of all registered client processes, update its local state and transmit a report (on a per process basis) to a number of specific external data collection daemons. These data collecting daemons provided local repositories that permitted knowledge of the availability of monitored components based on the received status reports. The daemons also recognised status changes and notified applications that registered an interest in a particular process.

The HBM was capable of process status monitoring and fault detection. HBM was unable to monitor resource performance, just availability, was based on a non-standard message format and required component daemons to be ported to all the available platforms used with a Grid.

3.2.2 NetLogger

The HBM was capable of process status monitoring and fault detection. HBM was unable to monitor resource performance, just availability, was based on a non-standard message format and required component daemons to be ported to all the available platforms used with a Grid. The NetLogger (Networked Application Logger) toolkit [2] provides tools for distributed application performance monitoring and analysis. The toolkit enables users to debug, tune and detect bottlenecks in distributed applications. NetLogger components include client libraries, data monitoring, storage, retrieval and visualisation tools, and an open messaging format.

The NetLogger messages can comprise of string, binary or XML encoded formats. The client library allows developers to add calls to existing source code so that monitoring events can be generated from their applications (events can be sent to file, network server, syslogd, or memory). The visualisation tool provides a means for event logs to be analysed. The storage and retrieval tools include a daemon to combine NetLogger events from multiple sources to a single central host and an event archive system.

NetLogger provides a means to debug, tune and detect bottlenecks in distributed applications with a common messaging format. NetLogger requires source code modification, there is a single data collection repository, so scalability will be an issue in a Grid environment, and it appears that there is no security within the system.

3.2.3 Autopilot

The Autopilot project, based on the Pablo Toolkit [3], is an infrastructure for real-time adaptive control of parallel and distributed computing resources. The goal of Autopilot is to support monitoring and steering of distributed applications. Autopilot allows application and runtime library developers to create software that can alter their behaviour and optimize performance in response to real-time external environment changes.

Autopilot control mechanisms automatically select and configure resource management algorithms based on observed system performance. Autopilot utilises distributed performance sensors to capture application and system performance data. Resource management policies are adapted in response to changes in the system environment. Sensor information is made available to clients in real-time, while actuators enable remote control of executing code and management policies.

Autopilot provides distributed application performance monitoring in a Grid environment. A Java GUI can be used to interact with remote sensors and actuators, and to present frequently measured host information.

3.2.4 Remos

The Remos (REsource MOnitoring System) [4] allows network-aware applications to obtain information about their distributed execution environment by providing an interface that hides system heterogeneity. Remos aims to balance information accuracy (best-effort or statistical information) with efficiency (providing a query-based interface), to manage monitoring overheads.

Remos consists of multiple (SNMP-based) collectors, which gather information about the network. Remos permits applications to obtain monitoring information in a portable manner, using platform independent interfaces, however it appears to lack any security mechanisms.

3.2.5 JAMM

Java Agents for Monitoring and Management (JAMM) [5] is a distributed set of components (sensors) that collect and publish monitoring data (events) about computer systems. JAMM agents securely automate the execution of monitoring sensors and the collection of event data.

Gateways allow clients to control or subscribe to a number of monitoring sensors, provide multiplexing/de-multiplexing and information filtering tasks. Loggers control the operation of sensors, and manage subscriptions for sensor output. Sensors execute on host systems, parse output from processes and transmit monitoring information to subscribers. A data collector combines data from multiple sensors into a single file for real-time visualization and analysis. A centralised directory service is required for event consumers to locate (registered) event producers.

JAMM uses a subscription-based, event notification approach for monitoring processes executing on hosts. The central directory service is implemented using a hierarchy of replicated LDAP servers. Java based sensors can be implemented as Activateable objects, permitting sensors to be added, removed dynamically or reconfigured, while non-Java sensors must be installed on each host.

With JAMM, it appears that sensors are key aspects of concern. If the sensor is pure Java then there is only a limited amount of real system data that can be collected without resorting to native system calls. In addition, sensor daemons need to be installed on every resource that is to be monitored or managed.

3.2.6 Summary of Related Work

Varieties of different systems exist for monitoring and managing distributed Grid-based resources and applications. Each system discussed in the section ante, has its own strengths and weaknesses. However, for resource monitoring and management, we believe that none fulfil the criteria that are desired in today's Grid-based environments. That of using standard and open components (Jini, Java applets, servlets, JDBC, and an SQL database), being scalable, requiring no additional modification or installation of additional software on the monitored resources (SNMP agents and MIBs), taking guidelines from the Grid and Services Oriented Architecture.

3.3 Technology Review

It was also imperative before the start of the project to search for the suitable technology to be adopted for the project. A detailed look over the different technologies was done and JINI was selected due to support for our basic architecture as shown below. A support for web services has also been provided in order to make the project compatible with the Globus Toolkit, currently the only Toolkit considered to implement GRID computing.

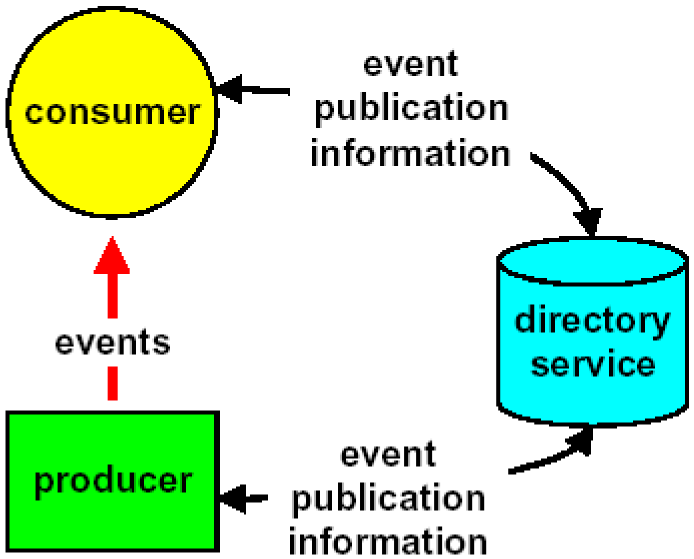


Figure 3-1 Basic Architecture of the Monitoring Service

A brief view of the technology is also being presented below.

3.3.1 JINI

It extends the Java Platform providing support for Distributed Applications. JINI is a set of Java classes (APIs) and services within a distributed computing framework.
The purpose of the JINI architecture is to federate groups of software components into a single, dynamic distributed system. The major features which make JINI technology attractive are:

3.3.1.1 Lookup Discovery Service

Services are found and resolved by a lookup service. The lookup service is the central bootstrapping mechanism for the system and provides the major point of contact between the system and users of the system.

3.3.1.2 Leasing Mechanism

Access to many of the services in the JINI system environment is lease based. A lease is a grant of guaranteed access over a time period. Each lease is negotiated between the user of the service and the provider of the service as part of the service protocol.

3.3.1.3 Transactions Manager

Reliable distributed object models require transaction support to aid in protecting the integrity of the resource layer. The specified transactions are inherited from the Jini programming model and focus on supporting large numbers of heterogeneous resources, rather than a single large resource (database). This service provides a series of operations, either within a single service or spanning multiple services that can be seen as distributed atomic procedures having a two-phase commit.

3.3.1.4 The JavaSpaces Service

This communication mechanism was heavily influenced by the concept of a Tuple space that was first described in 1982 in a programming language called Linda . Distributed programs, unaware of each other's existence, can communicate by releasing data (a tuple) into a persistent space. Programs read, write, and take such entries from the Tuple space that are of interest to them. The distributed JavaSpace implementation developed by Gigaspaces will be used for the data management system using the real-time monitoring services.

3.3.2 Web services

These are software components that can be invoked across a network using XML. They are building blocks for building distributed applications in language, platform and location independent way. A service-oriented architecture implements each part of the a systems as a web service. Simple web services provide low-level features such as access to particular kind of data or to perform a data processing task . Higher-level web services orchestrate lower-level services to provide more complex behaviors, resulting in a pyramid or increasingly specialized processing.

Three standards support the implementation of web services:

3.3.2.1 WSDL (web services description language)

It is the XML equivalent of a resume. WSDL describes what a web service can do, where it resides, and how to invoke it.

3.3.2.2 SOAP (service-oriented access protocol)

It is a general-purpose protocol for sending XML messages between endpoints, and may be used for remote procedure calls (RPC) or plain document transfer. SOAP messages can be send over any transport layer; HTTP is the most common layer, with implementations also available for Simple Mail Transport Protocol (SMTP) and Java Messaging Service (JMS).

3.3.2.3 UDDI (universal description, discovery and integration)

It is a registry for connecting producers and consumers of web services. A producer can use the UDDI publish API to register information about a web service, and a consumer can user the UDDI inquire API to locate one or more web services that satisfiews a particular criteria.

References

[1] http://www-fp.globus.org/hbm/heartbeat_spec.html, HearBeatMonitor Man

[2] http://www-didc.lbl.gov/NetLogger/, NetLogger Specifications

[3] http://ww.mit.edu:8001/people/mkgray/autopilot.html, AutoPolit Specifications

[4] http://www.remos.com/, REMOs Specifications

[5] http://www-didc.lbl.gov/JAMM/, JAMM Specifications

Chapter Four

SNMP – Simple Network Management Protocol

4.1 Introduction

The *Simple Network Management Protocol (SNMP)[1]* is an application layer protocol that facilitates the exchange of management information between network devices. It is part of the Transmission Control Protocol/Internet Protocol (TCP/IP)[2] protocol suite. SNMP [1] enables network administrators to manage network performance, find and solve network problems, and plan for network growth.

An SNMP-managed network consists of three key components: managed devices, agents, and network-management systems (NMSs). A *managed device* is a network node that contains an SNMP agent and that resides on a managed network. Managed devices collect and store management information and make this information available to NMSs using SNMP. Managed devices, sometimes called network elements, can be routers and access servers, switches and bridges, hubs, computer hosts, or printers.

An *agent* is a network-management software module that resides in a managed device. An agent has local knowledge of management information and translates that information into a form compatible with SNMP.

An *NMS* executes applications that monitor and control managed devices. NMSs provide the bulk of the processing and memory resources required for network management. One or more NMSs must exist on any managed network.

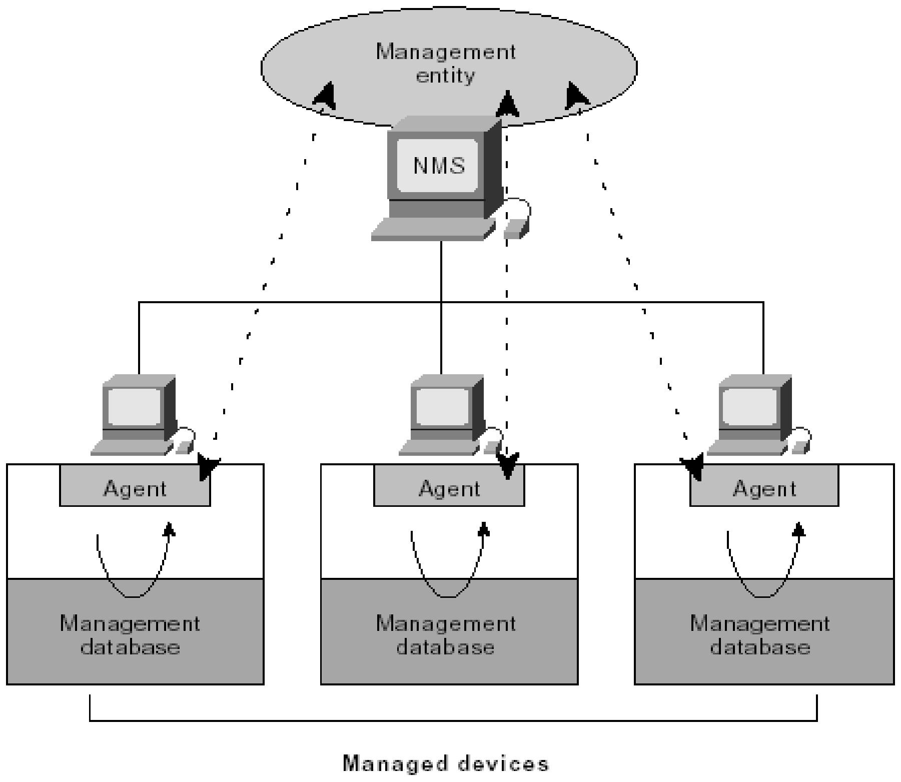


Figure 4-1 : SNMP centralized approach Model

4.2 SNMP and UDP

SNMP uses the *User Datagram Protocol* (UDP) as the transport protocol for passing data between managers and agents. UDP, defined in RFC 768, was chosen over the *Transmission Control Protocol* (TCP) because it is connectionless; that is, no end-to-end connection is made between the agent and the NMS when *datagrams* (packets) are sent back and forth. This aspect of UDP makes it unreliable, since there is no acknowledgment of lost datagrams at the protocol level. It's up to the SNMP application to determine if datagrams are lost and retransmit them if it so desires. This is typically accomplished with a simple timeout. The NMS sends a UDP request to an agent and waits for a response. The length of time the NMS waits depends on how it's configured. If the timeout is reached and the NMS has not heard back from the agent, it assumes the packet was lost and retransmits the request. The number of times the NMS retransmits packets is also configurable.

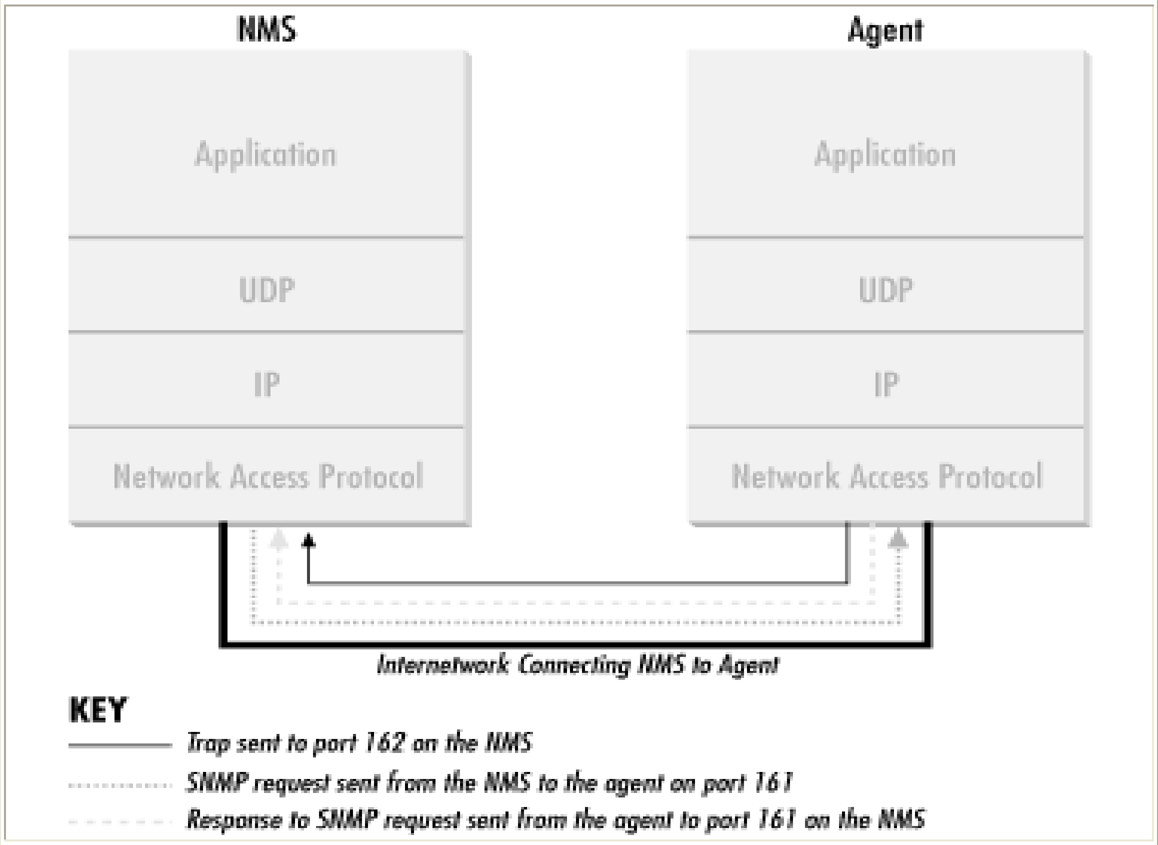
At least as far as regular information requests are concerned, the unreliable nature of UDP isn't a real problem. At worst, the management station issues a request and never receives a response. For traps, the situation is somewhat different. If an agent sends a trap and the trap never arrives, the NMS has no way of knowing that it was ever sent. The agent doesn't even know that it needs to resend the trap, because the NMS is not required to send a response back to the agent acknowledging receipt of the trap.

The upside to the unreliable nature of UDP is that it requires low overhead, so the impact on network's performance is reduced. SNMP has been implemented over TCP, but this is more for special-case situations in which someone is developing an agent for a proprietary piece of equipment. In a heavily congested and managed network, SNMP over TCP is a bad idea. It's also worth realizing that TCP isn't magic, and that SNMP is designed for working with networks that are in trouble--if your network never failed, you wouldn't need to monitor it. When a network is failing, a protocol that tries to get the data through but gives up if it can't is almost certainly a better design choice than a protocol that will flood the network with retransmissions in its attempt to achieve reliability.

SNMP uses the UDP port 161 for sending and receiving requests, and port 162 for receiving traps from managed devices. Every device that implements SNMP must use these port numbers as the defaults, but some vendors allow you to change the default ports in the agent's configuration. If these defaults are changed, the NMS must be made aware of the changes so it can query the device on the correct ports.

Figure 4-2 shows the TCP/IP protocol suite, which is the basis for all TCP/IP communication. Today, any device that wishes to communicate on the Internet (e.g., Windows NT systems, Unix servers, Cisco routers, etc.) must use this protocol suite. This model is often referred to as a protocol stack, since each layer uses the information from the layer directly below it and provides a service to the layer directly above it.

Figure 4-2 : **TCP/IP communication model and SNMP**


NMS
Agent
Application
UDP
IP
Network Access Protocol
Application
UDP
IP
Network Access Protocol
Internetwork Connecting NMS to Agent
KEY
Trap sent to port 162 on the NMS
SNMP request sent from the NMS to the agent on port 161
Response to SNMP request sent from the agent to port 161 on the NMS

When either an NMS or an agent wishes to perform an SNMP function (e.g., a request or trap), the following events occur in the protocol stack:

4.2.1 Application

First, the actual SNMP application (NMS or agent) decides what it's going to do. For example, it can send an SNMP request to an agent, send a response to an SNMP request (this would be sent from the agent), or send a trap to an NMS. The application layer provides services to an end user, such as an operator requesting status information for a port on an Ethernet switch.

4.2.2 UDP

The next layer, UDP [3], allows two hosts to communicate with one another. The UDP header contains, among other things, the destination port of the device to which it's sending the request or trap. The destination port will either be 161 (query) or 162 (trap).

4.2.3 IP

The IP layer tries to deliver the SNMP packet to its intended destination, as specified by its IP address.

4.2.4 Medium Access Control (MAC)

The final event that must occur for an SNMP packet to reach its destination is for it to be handed off to the physical network, where it can be routed to its final destination. The MAC layer is comprised of the actual hardware and device drivers that put the data onto a physical piece of wire, such as an Ethernet card. The MAC layer also is responsible for receiving packets from the physical network and sending them back up the protocol stack so they can be processed by the application layer (SNMP, in this case).

4.3 SNMP Communities

SNMP uses the notion of communities to establish trust between managers and agents. An agent is configured with three community names: read-only, read-write, and trap. The community names are essentially passwords; there's no real difference between a community string and the password used to access a computer. The three community strings control different kinds of activities. As its name implies, the read-only community string only lets read data values, but doesn't allow a modification to it. For example, it allows to read the number of packets that have been transferred through the ports on router, but doesn't allow to reset the counters. The read-write community is allowed to read and modify data values; with the read-write community string, you can read the counters, reset their values, and even reset the interfaces or do other things that change the router's configuration. Finally, the trap community string allows you to receive traps (asynchronous notifications) from the agent.

Most vendors ship their equipment with default community strings, typically *public* for the read-only community and *private* for the read-write community. It's important to change these defaults before the device goes live on the network. When setting up an SNMP agent, one may want to configure its trap destination, which is the address to which it will send any traps it generates. In addition, since SNMP community strings are sent in clear text, therefore authentication-failure traps can also be configured, in case of illegal access attempts, to be sent to the network administrator. Among other things, authentication-failure traps can be very useful in determining when an intruder might be trying to gain access to the network.

It is important to realize that if someone has read-write access to any of SNMP devices, he can gain control of those devices by using SNMP (for example, he can set router interfaces, switch ports down, or even modify your routing tables). One way to protect community strings is to use a *Virtual Private Network* (VPN) to make sure the network traffic is encrypted. Another way is to change the community strings often. Changing community strings isn't difficult for a small network, but for a network that spans city blocks or more and has dozens (or hundreds or thousands) of managed hosts, changing community strings can be a problem. An easy solution is to write a simple Perl script that uses SNMP to change the community strings on your devices.

4.4 The Structure of Management Information

The first step toward understanding what kind of information a device can provide is to understand how this data itself is represented within the context of SNMP. The *Structure of Management Information* defines precisely how managed objects are named and specifies their associated datatypes. A managed object (sometimes called a MIB object, an object, or a MIB) is one of any number of specific characteristics of a managed device. Managed objects are comprised of one or more object instances, which are essentially variables. Two types of managed objects exist: scalar and tabular. *Scalar objects* define a single object instance. *Tabular objects* define multiple related object instances that are grouped in MIB tables.

The definition of managed objects can be broken down into three attributes:

4.4.1 Name

The name, or *object identifier* (OID), uniquely defines a managed object. Names commonly appear in two forms: numeric and "human readable." In either case, the names are long and inconvenient. In SNMP applications, a lot of work goes into helping you navigate through the namespace conveniently.

4.4.2 Type and syntax

A managed object's datatype is defined using a subset of *Abstract Syntax Notation One* (ASN.1). ASN.1 is a way of specifying how data is represented and transmitted between managers and agents, within the context of SNMP. The nice thing about ASN.1 is that the notation is machine-independent. This means that a PC running Windows NT can communicate with a Sun SPARC machine and not have to worry about things such as byte ordering.

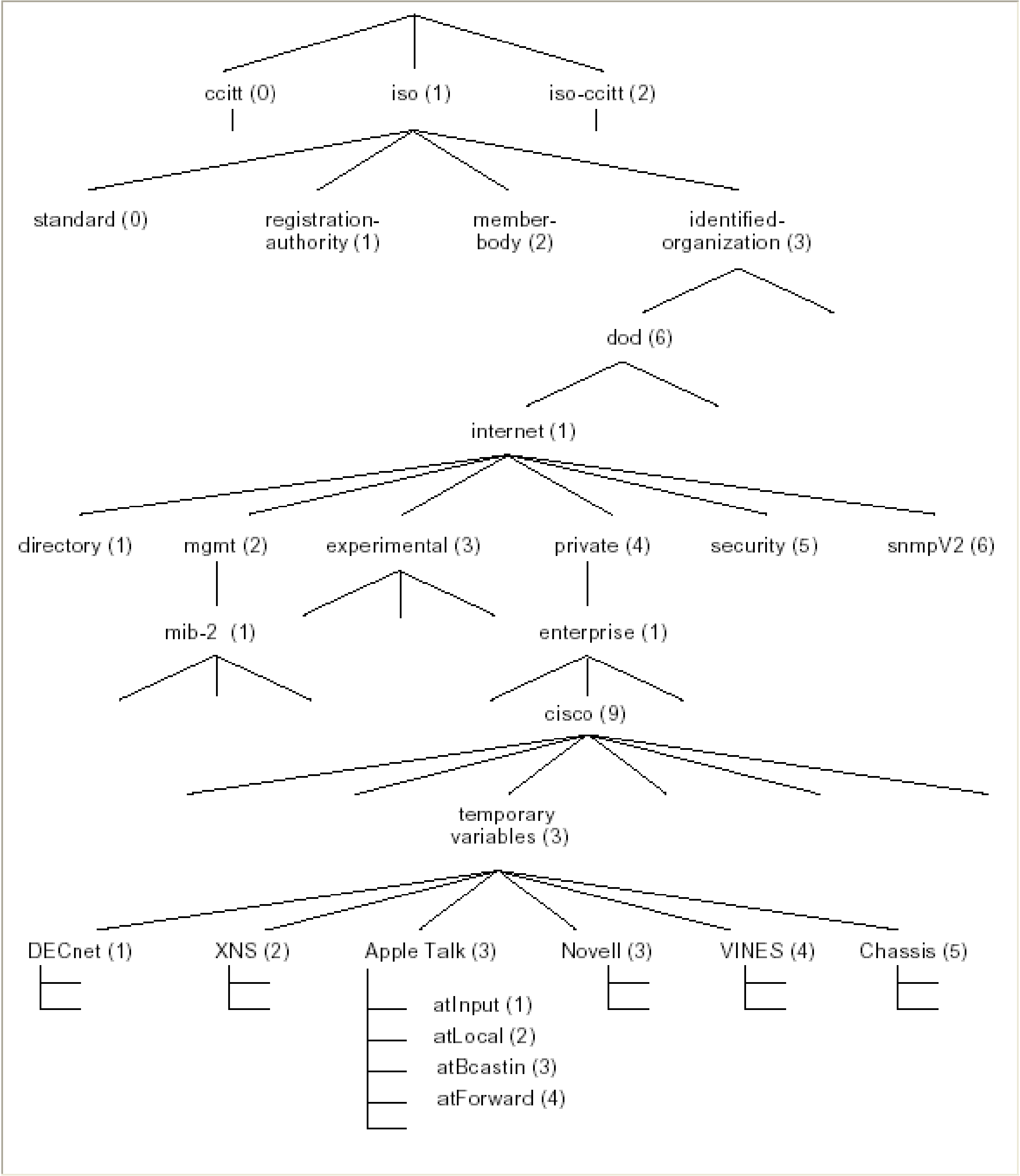
4.4.3 Encoding

A single instance of a managed object is encoded into a string of octets using the *Basic Encoding Rules* (BER). BER defines how the objects are encoded and decoded so they can be transmitted over a transport medium such as Ethernet.

4.5 Naming OIDs

Managed objects are organized into a tree-like hierarchy. This structure is the basis for SNMP's naming scheme. An object ID is made up of a series of integers based on the nodes in the tree, separated by dots (.). Although there's a human-readable form that's more friendly than a string of numbers, this form is nothing more than a series of names separated by dots, each of which represents a node of the tree. Therefore both sequence of numbers and names can be used

Figure 4-3. SMI object tree



In the object tree, the node at the top of the tree is called the *root*, anything with children is called a *subtree*, and anything without children is called a *leaf node*. For example, Figure 4-3's root, the starting point for the tree, is called "Root-Node." Its subtree is made up of *ccitt(0)*, *iso(1)*, and iso-ccit*(2).* In this illustration, *iso(1)* is the only node that contains a subtree; the other two nodes are both leaf nodes.

Each managed object has a numerical OID and an associated textual name. The dotted-decimal notation is how a managed object is represented internally within an agent; the textual name, like an IP domain name, saves humans from having to remember long, tedious strings of integers.

The *directory* branch currently is not used. The *management* branch, or *mgmt*, defines a standard set of Internet management objects. The *experimental* branch is reserved for testing and research purposes. Objects under the *private* branch are defined unilaterally, which means that individuals and organizations are responsible for defining the objects under this branch. Here is the definition of the *internet* subtree, as well as all four of its subtrees:

```
internet      OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 }
directory     OBJECT IDENTIFIER ::= { internet 1 }
mgmt          OBJECT IDENTIFIER ::= { internet 2 }
experimental  OBJECT IDENTIFIER ::= { internet 3 }
private       OBJECT IDENTIFIER ::= { internet 4 }
```

The first line declares *internet* as the OID *1.3.6.1*, which is defined as a subtree of *iso.org.dod*, or *1.3.6* (the ::= is a definition operator). The last four declarations are similar, but they define the other branches that belong to *internet*. For the *directory* branch, the notation { internet 1 } tells us that it is part of the *internet* subtree, and that its OID is *1.3.6.1.1*. The OID for *mgmt* is *1.3.6.1.2*, and so on.

There is currently one branch under the *private* subtree. It's used to give hardware and software vendors the ability to define their own private objects for any type of hardware or software they want managed by SNMP. Its SMI definition is:

```
enterprises   OBJECT IDENTIFIER ::= { private 1 }
```

The Internet Assigned Numbers Authority (IANA) currently manages all the private enterprise number assignments for individuals, institutions, organizations, companies, etc A list of all the current private enterprise numbers can be obtained from ftp://ftp.isi.edu/in-notes/iana/assignments/enterprise-numbers. As an example, Cisco Systems's private enterprise number is 9, so the base OID for its private object space is defined as *.1.3.6.1.4.1.9 or* i*so.org.dod.internet.private.enterprises.cisco*. Cisco is free to do as it wishes with this private branch. It's typical for companies such as Cisco that manufacture networking equipment to define their own private enterprise objects. This allows for a richer set of management information than can be gathered from the standard set of managed objects defined under the *mgmt* branch.

Companies aren't the only ones who can register their own private enterprise numbers. Anyone can do so, and it's free. The web-based form for registering private enterprise numbers can be found at http://www.isi.edu/cgi-bin/iana/enterprise.pl.

4.6 Defining OIDs

The SYNTAX attribute provides for definitions of managed objects through a subset of ASN.1. SMI defines several datatypes that are paramount to the management of networks and network devices. It's important to keep in mind that these datatypes are simply a way to define what kind of information a managed object can hold. The types are similar to those that are found in a computer programming language like C. Table 2-1 lists the supported datatypes for SMI.

Table 4-1: SMI Datatypes

Datatype	Description
INTEGER	A 32-bit number often used to specify enumerated types within the context of a single managed object. For example, the operational status of a router interface can be *up*, *down*, or *testing*. With enumerated types, 1 would represent up, 2 down, and 3 testing. The value zero (0) must not be used as an enumerated type, according to RFC 1155.
OCTET STRING	A string of zero or more octets (more commonly known as

	bytes) generally used to represent text strings, but also sometimes used to represent physical addresses.
Counter	A 32-bit number with minimum value 0 and maximum value *232* - 1 (4,294,967,295). When the maximum value is reached, it wraps back to zero and starts over. It's primarily used to track information such as the number of octets sent and received on an interface or the number of errors and discards seen on an interface. A Counter is monotonically increasing, in that its values should never decrease during normal operation. When an agent is rebooted, all Counter values should be set to zero. Deltas are used to determine if anything useful can be said for successive queries of Counter values. A delta is computed by querying a Counter at least twice in a row, and taking the difference between the query results over some time interval.
OBJECT IDENTIFIER	A dotted-decimal string that represents a managed object within the object tree. For example, *1.3.6.1.4.1.9* represents Cisco Systems's private enterprise OID.
NULL	Not currently used in SNMP.
SEQUENCE	Defines lists that contain zero or more other ASN.1 datatypes.
SEQUENCE OF	Defines a managed object that is made up of a SEQUENCE of ASN.1 types.
IpAddress	Represents a 32-bit IPv4 address. Neither SMIv1 nor SMIv2 discusses 128-bit IPv6 addresses; this problem will be addressed by the IETF's SMI Next Generation (SMING) working group (see http://www.ietf.org/html.charters/sming-charter.html).
NetworkAddress	Same as the IpAddress type, but can represent different network address types.
Gauge	A 32-bit number with minimum value 0 and maximum value *232* - 1 (4,294,967,295). Unlike a Counter, a Gauge can increase and decrease at will, but it can never exceed its maximum value. The interface speed on a router is measured with a Gauge.
TimeTicks	A 32-bit number with minimum value 0 and maximum value *232* - 1 (4,294,967,295). TimeTicks measures time in hundredths of a second. Uptime on a device is measured using this datatype.
Opaque	Allows any other ASN.1 encoding to be stuffed into an OCTET STRING.
Integer32	Same as an INTEGER.
Counter32	Same as a Counter.
Gauge32	Same as a Gauge.

Unsigned32	Represents decimal values in the range of 0 to *232* - 1 inclusive.
Counter64	Similar to Counter32, but its maximum value is 18,446,744,073,709,551,615. Counter64 is ideal for situations in which a Counter32 may wrap back to 0 in a short amount of time.
BITS	An enumeration of nonnegative named bits.

The goal of all these object types is to define managed objects. MIB is a logical grouping of managed objects as they pertain to a specific management task, vendor, etc. The MIB can be thought of as a specification that defines the managed objects a vendor or device supports. Vendor-specific MIBs typically are distributed as human-readable text files that can be inspected (or even modified) with a standard text editor such as *vi*.

Most modern NMS products maintain a compact form of all the MIBs that define the set of managed objects for all the different types of devices they're responsible for managing. NMS administrators will typically compile a vendor's MIB into a format the NMS can use. Once a MIB has been loaded or compiled, administrators can refer to managed objects using either the numeric or human-readable object ID.

4.7 MIB-II – Mostly Used Group in this project

MIB-II is a very important management group, because every device that supports SNMP must also support MIB-II. Most of the objects from this group have been used in my project, I therefore would elaborate it, a bit. The section of *RFC1213-MIB* that defines the base OIDs for the *mib-2* subtree looks like this:

```
mib-2        OBJECT IDENTIFIER ::= { mgmt 1 }
system       OBJECT IDENTIFIER ::= { mib-2 1 }
interfaces   OBJECT IDENTIFIER ::= { mib-2 2 }
at           OBJECT IDENTIFIER ::= { mib-2 3 }
ip           OBJECT IDENTIFIER ::= { mib-2 4 }
icmp         OBJECT IDENTIFIER ::= { mib-2 5 }
tcp          OBJECT IDENTIFIER ::= { mib-2 6 }
```

```
udp         OBJECT IDENTIFIER ::= { mib-2 7 }
egp         OBJECT IDENTIFIER ::= { mib-2 8 }
transmission OBJECT IDENTIFIER ::= { mib-2 10 }
snmp         OBJECT IDENTIFIER ::= { mib-2 11 }
```

mib-2 is defined as *iso.org.dod.internet.mgmt.1*, or *1.3.6.1.2.1*. From here, we can see that the *system* group is mib-2 1, or *1.3.6.1.2.1.1*, and so on. Figure 2-4 shows the MIB-II subtree of the *mgmt* branch.

Figure 4-4. MIB-II subtree

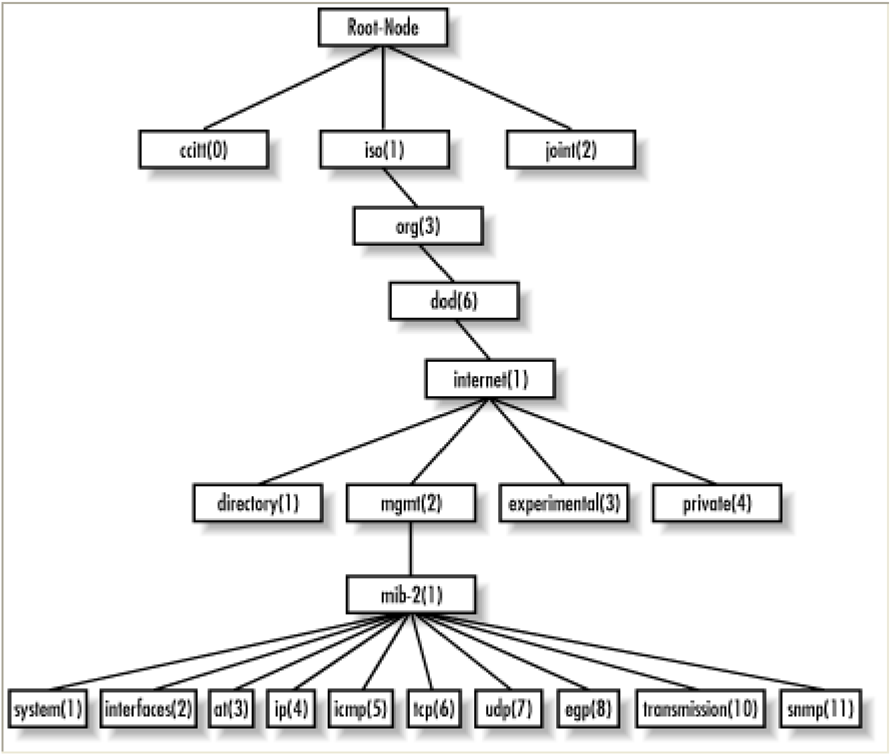


Table 4-2 briefly describes each of the management groups defined in MIB-II. We don't go into great detail about each group, since you can pull down RFC 1213 and read the MIB yourself.

Table 4-2: Brief Description of the MIB-II Groups

Subtree Name	OID	Description
System	*1.3.6.1.2.1.1*	Defines a list of objects that pertain to system operation, such as the system uptime, system contact, and system name.
interfaces	*1.3.6.1.2.1.2*	Keeps track of the status of each interface on a

		managed entity. The *interfaces* group monitors which interfaces are up or down and tracks such things as octets sent and received, errors and discards, etc.
At	*1.3.6.1.2.1.3*	The address translation (*at*) group is deprecated and is provided only for backward compatibility. It will probably be dropped from MIB-III.
Ip	*1.3.6.1.2.1.4*	Keeps track of many aspects of IP, including IP routing.
icmp	*1.3.6.1.2.1.5*	Tracks things such as ICMP errors, discards, etc.
Tcp	*1.3.6.1.2.1.6*	Tracks, among other things, the state of the TCP connection (e.g., *closed*, *listen*, *synSent*, etc.).
udp	*1.3.6.1.2.1.7*	Tracks UDP statistics, datagrams in and out, etc.
Egp	*1.3.6.1.2.1.8*	Tracks various statistics about EGP and keeps an EGP neighbor table.
transmission	*1.3.6.1.2.1.10*	There are currently no objects defined for this group, but other media-specific MIBs are defined using this subtree.
snmp	*1.3.6.1.2.1.11*	Measures the performance of the underlying SNMP implementation on the managed entity and tracks things such as the number of SNMP packets sent and received.

4.8 SNMP Operations

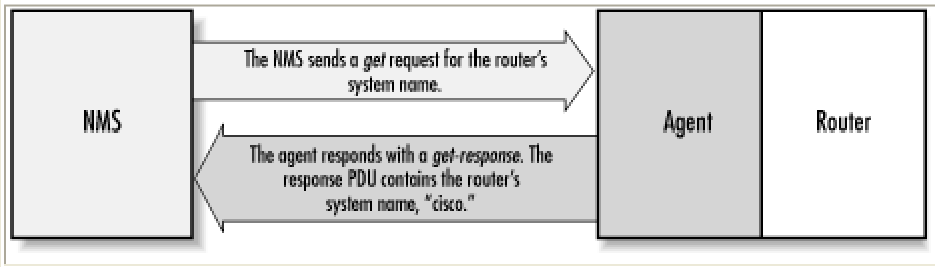
These operations are about the gathering of information, from the SNMP objects managing this information.

The *Protocol Data Unit* (PDU) is the message format that managers and agents use to send and receive information. There is a standard PDU format for each of the following SNMP operations: These formats shall be presented in the annex “PDU Formats”.

4.8.1 The get Operation

The *get* request is initiated by the NMS, which sends the request to the agent. The agent receives the request and processes it to best of its ability. Some devices that are under heavy load, such as routers, may not be able to respond to the request and will have to drop it. If the agent is successful in gathering the requested information, it sends a get-response back to the NMS, where it is processed. This process is illustrated in Figure 2-5.

Figure 4-5. get request sequence


NMS
The NMS sends a *get* request for the router's system name.
The agent responds with a *get-response*. The response PDU contains the router's system name, "cisco."
Agent
Router

How did the agent know what the NMS was looking for? One of the items in the get request is a *variable binding*. A variable binding, or varbind, is a list of MIB objects that allows a request's recipient to see what the originator wants to know. Variable bindings can be thought of as *OID=value* pairs that make it easy for the originator (the NMS, in this case) to pick out the information it needs when the recipient fills the request and sends back a response. Let's look at this operation in action:

```
snmpget afras.research.niit.edu.pk –c public .1.3.6.1.2.1.1.6.0
system.sysLocation.0 = ""
```

This command “snmpget” is firstly specifying the host (i.e. managed device) from which we wish to retrieve the information, then “–c” switch is for community string which in this example is “public” and then the oid. “system.sysLocation.0=”” “ is the result of the command which shows a null string.

The get command is useful for retrieving a single MIB object at a time. Trying to manage anything in this manner can be a waste of time, though. This is where the get-next command comes in. It allows you to retrieve more than one object from a device, over a period of time.

4.8.2 The get-next Operation

The get-next operation lets you issue a sequence of commands to retrieve a group of values from a MIB. In other words, for each MIB object we want to retrieve, a separate get-next request and get-response are generated. The get-next command traverses a subtree in lexicographic order. Since an OID is a sequence of integers, it's easy for an agent to start at the root of its SMI object tree and work its way down until it finds the OID it is looking for. When the NMS receives a response from the agent for the get-next command it just issued, it issues another get-next command. It keeps doing this until the agent returns an error, signifying that the end of the MIB has been reached and there are no more objects left to get.

If we look at another example, we can see this behavior in action. This time we'll use a command called snmpwalk. This command simply facilitates the get-next procedure for us. It's invoked just like the snmpget command, except this time we specify which branch to start at (in this case, the *system* group):

```
snmpwalk mlin.research.niit.edu.pk –c public system
system.sysDescr.0 = "Cisco Internetwork Operating System Software
..IOS (tm) 2500 Software (C2500-I-L), Version 11.2(5), RELEASE
SOFTWARE (fc1)..Copyright (c) 1986-1997 by cisco Systems, Inc...
Compiled Mon 31-Mar-97 19:53 by ckralik"
system.sysObjectID.0 = OID: enterprises.9.1.19
system.sysUpTime.0 = Timeticks: (27210723) 3 days, 3:35:07.23
system.sysContact.0 = "administrator@niit.edu.pk"
system.sysName.0 = "mlin"
system.sysLocation.0 = "Rawalpindi, Pakistan"
system.sysServices.0 = 6
```

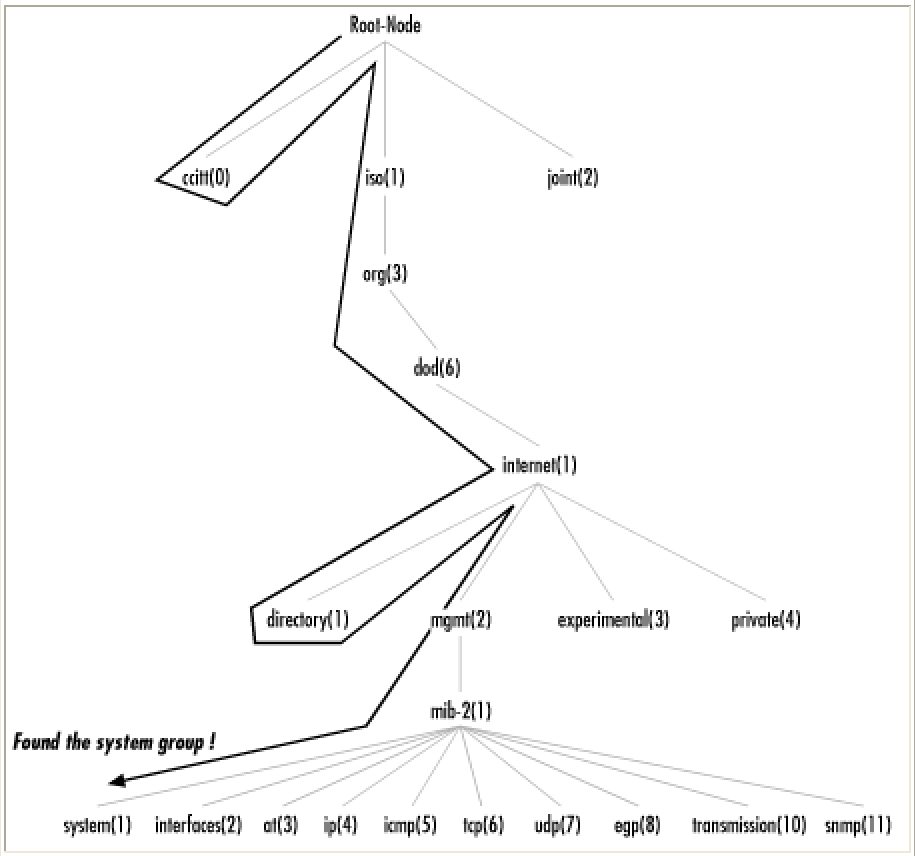
The get-next sequence returns seven MIB variables. Each of these objects is part of the *system* group as it's defined in RFC 1213. We see a system object ID, the amount of time the system has been up, the contact person, etc.
Given that you've just looked up some object, how does get-next figure out which object to look up next? get-next is based on the concept of the lexicographic ordering of the MIB's object tree. This order is made much simpler because every

node in the tree is assigned a number. To understand what this means, let's start at the root of the tree and walk down to the *system* node.

To get to the *system* group (OID *1.3.6.1.2.1.1*), we start at the root of the object tree and work our way down. Figure 4-6 shows the logical progression from the root of the tree all the way to the *system* group. At each node in the tree, we visit the lowest-numbered branch. Thus, when we're at the root node, we start by visiting *ccitt*. This node has no nodes underneath it, so we move to the *iso* node. Since *iso* does have a child we move to that node, *org*. The process continues until we reach the *system* node. Since each branch is made up of ascending integers (*ccitt(0) iso(1) join(2)*, for example), the agent has no problem traversing this tree structure all the way down to the *system(1)* group. If we were to continue this walk, we'd proceed to *system.1* (*system.sysLocation*), *system.2*, and the other objects in the *system* group. Next, we'd go to *interfaces(2)*, and so on.

Figure 4-3. SMI object tree

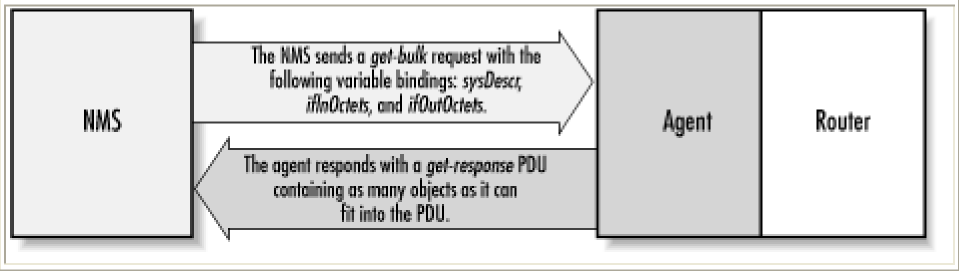
Figure 4-6. Walking the MIB tree



4.8.3 The get-bulk Operation

SNMPv2 defines the get-bulk operation, which allows a management application to retrieve a large section of a table at once. The standard get operation can attempt to retrieve more than one MIB object at once, but message sizes are limited by the agent's capabilities. If the agent can't return all the requested responses, it returns an error message with no data. The get-bulk operation, on the other hand, tells the agent to send as much of the response back as it can. This means that incomplete responses are possible. Two fields must be set when issuing a get-bulk command: nonrepeaters and max-repetitions. Nonrepeaters tells the get-bulk command that the first *N* objects can be retrieved with a simple get-next operation. Max-repetitions tells the get-bulk command to attempt up to *M* get-next operations to retrieve the remaining objects. Figure 2-7 shows the get-bulk command sequence.

Figure 4-7. get-bulk request sequence


NMS
The NMS sends a *get-bulk* request with the following variable bindings: *sysDescr*, *ifInOctets*, and *ifOutOctets*.
The agent responds with a *get-response* PDU containing as many objects as it can fit into the PDU.
Agent
Router

In Figure 4-7, we're requesting three bindings: *sysDescr*, *ifInOctets*, and *ifOutOctets*. The total number of variable bindings that we've requested is given by the formula *N + (M * R)*, where *N* is the number of nonrepeaters (i.e., scalar objects in the request--in this case 1, because *sysDescr* is the only scalar object), *M* is max-repetitions (in this case, we've set it arbitrarily to 3), and *R* is the number of nonscalar objects in the request (in this case 2, because *ifInOctets* and *ifOutOctets* are both nonscalar). Plugging in the numbers from this example, we get 1 + (3 * 2) = 7, which is the total number of variable bindings that can be returned by this get-bulk request.

The Net-SNMP package comes with a command for issuing get-bulk queries. If we execute this command using all the parameters previously discussed, it will look like the following:

```
snmpbulkget -v2c -B 1 3 afras.research.niit.edu.pk –c public sysDescr ifInOctets ifOutOctets
system.sysDescr.0 = "Linux linux 2.2.5-15 #3 Sat Apr 26 19:33:18 EDT 2003 i686"
interfaces.ifTable.ifEntry.ifInOctets.1 = 70840
interfaces.ifTable.ifEntry.ifOutOctets.1 = 70840
interfaces.ifTable.ifEntry.ifInOctets.2 = 143548020
interfaces.ifTable.ifEntry.ifOutOctets.2 = 111725152
interfaces.ifTable.ifEntry.ifInOctets.3 = 0
interfaces.ifTable.ifEntry.ifOutOctets.3 = 0
```

Since get-bulk is an SNMPv2 command, you have to tell snmpgetbulk to use an SNMPv2 PDU with the -v2c option. The nonrepeaters and max-repetitions are set with the -B 1 3 option. This sets nonrepeaters to 1 and max-repetitions to 3. Notice that the command returned seven variable bindings: one for *sysDescr* and three each for *ifInOctets* and *ifOutOctets*.

4.8.4 The set Operation

The *set* command is used to change the value of a managed object or to create a new row in a table. Objects that are defined in the MIB as read-write or write-only can be altered or created using this command. It is possible for an NMS to set more than one object at a time.

Figure 4-8. set request sequence

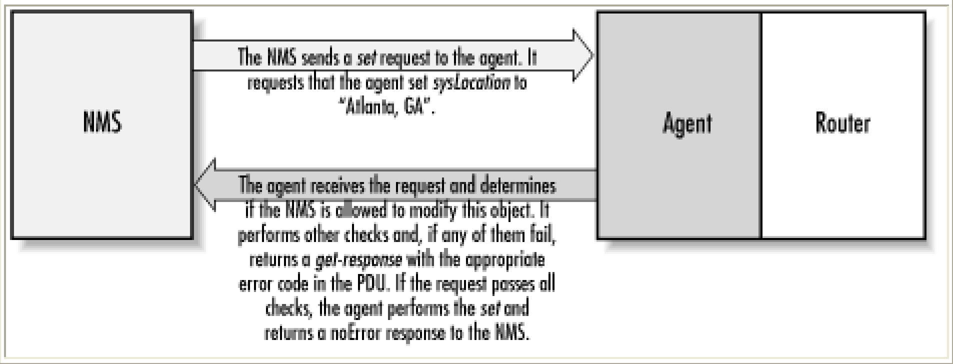

NMS
The NMS sends a *set* request to the agent. It requests that the agent set *sysLocation* to "Atlanta, GA".
Agent
Router
The agent receives the request and determines if the NMS is allowed to modify this object. It performs other checks and, if any of them fail, returns a *get-response* with the appropriate error code in the PDU. If the request passes all checks, the agent performs the *set* and returns a noError response to the NMS.

Figure 4-8 shows the set request sequence. It's similar to the other commands, but it is actually changing something in the device's configuration, as opposed to just retrieving a response to a query. If we look at an example of an actual set, you will see the command take place. The following example queries the *sysLocation* variable, then sets it to a value:

```
snmpget afras.research.niit.edu.pk –c public system.sysLocation.0
system.sysLocation.0 = ""

snmpset afras.research.niit.edu.pk private system.sysLocation.0 s "Rawalpindi,
Pakistan"
system.sysLocation.0 = "Rawalpindi, Pakistan"

snmpget afras.research.niit.edu.pk public system.sysLocation.0
system.sysLocation.0 = "Rawalpindi, Pakistan"
```

The first command is the get command, which displays the current value of *sysLocation*. Result is a null string. Now second command is given to set this string through snmpset. For this command, we supply the hostname, the read-write community string (*private*), and the variable we want to set (*system.sysLocation.0*), together with its new value (s "Rawalpindi, Pakistan"). The s tells snmpset that we want to set the value of *sysLocation* to a string; and "Rawalpindi, Pakistan" is the new value itself. How do we know that *sysLocation* requires a string value? The definition of *sysLocation* in RFC 1213 looks like this:

```
sysLocation OBJECT-TYPE
   SYNTAX  DisplayString (SIZE (0..255))
   ACCESS  read-write
   STATUS  mandatory
   DESCRIPTION
      "The physical location of this node (e.g., 'telephone closet,
       3rd floor')."
   ::= { system 6 }
```

The SYNTAX for *sysLocation* is DisplayString (SIZE (0..255)), which means that it's a string with a maximum length of 255 characters. The snmpset command succeeds and reports the new value of *sysLocation*. But just to confirm, we run a final snmpget, which tells us that the set actually took effect. It is possible to set more than one object at a time, but if any of the sets fail, they all fail (i.e., no values are changed). This behavior is intended.

get, get-next, get-bulk, and set Error Responses

Error responses help to determine whether get or set request was processed correctly by the agent. The get, get-next, and set operations can return the error responses shown in Table 4-3. The error status for each error is show in parentheses.

Table 4-3: SNMPv1 Error Messages

SNMPv1 Error Message	**Description**
noError(0)	There was no problem performing the request.
tooBig(1)	The response to your request was too big to fit into one response.
noSuchName(2)	An agent was asked to get or set an OID that it can't find; i.e., the OID doesn't exist.
badValue(3)	A read-write or write-only object was set to an inconsistent value.
readOnly(4)	This error is generally not used. The noSuchName error is equivalent to this one.
genErr(5)	This is a catch-all error. If an error occurs for which none of the previous messages is appropriate, a genError is issued.

The SNMPv1 error messages are not very robust. In an attempt to fix this problem, SNMPv2 defines additional error responses that are valid for get, set, get-next, and get-bulk operations, provided that both the agent and NMS support SNMPv2. These responses are listed in Table 4-4.

Table 4-4: SNMPv2 Error Messages

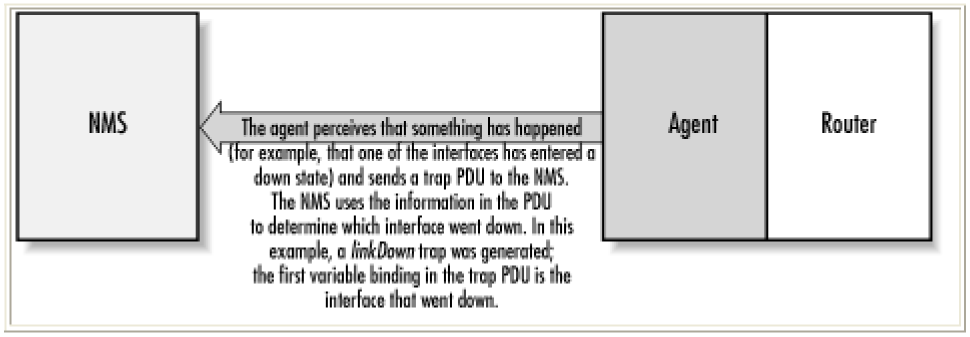
SNMPv2 Error Message	**Description**
noAccess(6)	A set to an inaccessible variable was attempted. This typically occurs when the variable has an ACCESS type of not-accessible.
wrongType(7)	An object was set to a type that is different from its

	definition. This error will occur if you try to set an object that is of type INTEGER to a string, for example.
wrongLength(8)	An object's value was set to something other than what it calls for. For instance, a string can be defined to have a maximum character size. This error occurs if you try to set a string object to a value that exceeds its maximum length.
wrongEncoding(9)	A set operation was attempted using the wrong encoding for the object being set.
wrongValue(10)	A variable was set to a value it doesn't understand. This can occur when a read-write is defined as an enumeration, and you try to set it to a value that is not one of the enumerated types.
noCreation(11)	You tried to set a nonexistent variable or create a variable that doesn't exist in the MIB.
inconsistentValue	A MIB variable is in an inconsistent state, and is not accepting any set requests.
resourceUnavailable(13)	No system resources are available to perform a set.
commitFailed(14)	This is a catch-all error for set failures.
undoFailed(15)	A set failed and the agent was unable to roll back all the previous sets up until the point of failure.
authorizationError(16)	An SNMP command could not be authenticated; in other words, someone has supplied an incorrect community string.
notWritable(17)	A variable will not accept a set, even though it is supposed to.
inconsistentName(18)	You attempted to set a variable, but that attempt failed because the variable was in some kind of inconsistent state.

4.9 SNMP Traps

A trap is a way for an agent to tell the NMS that something bad has happened. Figure 2-9 shows the trap-generation sequence.

Figure 4-9. Trap generation


NMS
The agent perceives that something has happened (for example, that one of the interfaces has entered a down state) and sends a trap PDU to the NMS. The NMS uses the information in the PDU to determine which interface went down. In this example, a *linkDown* trap was generated; the first variable binding in the trap PDU is the interface that went down.
Agent
Router

The trap originates from the agent and is sent to the trap destination, as configured within the agent itself. The trap destination is typically the IP address of the NMS. No acknowledgment is sent from the NMS to the agent, so the agent has no way of knowing if the trap makes it to the NMS. Since SNMP uses UDP, and since traps are designed to report problems with your network, traps are especially prone to getting lost and not making it to their destinations. However, the fact that traps can get lost doesn't make them any less useful; in a well-planned environment, they are an integral part of network management. It's better for your equipment to try to tell you that something is wrong, even if the message may never reach you, than simply to give up and let you guess what happened. Here are a few situations that a trap might report:

- A network interface on the device (where the agent is running) has gone down.
- A network interface on the device (where the agent is running) has come back up.
- An incoming call to a modem rack was unable to establish a connection to a modem.
- The fan on a switch or router has failed.

When an NMS receives a trap, it needs to know how to interpret it; that is, it needs to know what the trap means and how to interpret the information it carries. A trap is first identified by its generic trap number. There are seven generic trap numbers (0-6), shown in Table 4-5. Generic trap 6 is a special catch-all category for "enterprise-specific" traps, which are traps defined by vendors or users that fall outside of the six generic trap categories. Enterprise-specific traps are further identified by an enterprise ID (i.e., an object ID somewhere in the *enterprises* branch of the MIB tree, *iso.org.dod.internet.private.enterprises*) and a specific trap number chosen by the enterprise that defined the trap. Thus, the object ID of an enterprise-specific trap is *enterprise-id.specific-trap-number*. For example, when Cisco defines special traps for its private MIBs, it places them all in its enterprise-specific MIB tree (*iso.org.dod.internet.private.enterprises.cisco*). Any organization is free to define its own enterprise-specific traps; the only requirement is that they should register own enterprise number with IANA.

A trap is usually packed with information. This information is in the form of MIB objects and their values; as mentioned earlier, these object-value pairs are known as variable bindings. For the generic traps 0 through 5, knowledge of what the trap contains is generally built into the NMS software or trap receiver. The variable bindings contained by an enterprise-specific trap are determined by whosoever defined the trap. For example, if a modem in a modem rack fails, the rack's agent may send a trap to the NMS informing it of the failure. The trap will most likely be an enterprise-specific trap defined by the rack's manufacturer; the trap's contents are up to the manufacturer, but it will probably contain enough information to let you determine exactly what failed (for example, the position of the modem card in the rack and the channel on the modem card).

Table 4-5: Generic Traps

Generic Trap Name and Number	**Definition**
coldStart (0)	Indicates that the agent has rebooted. All management variables will be reset; specifically, Counters and Gauges will be reset to zero (0). One nice thing about the *coldStart* trap is that it can be used to determine when new hardware is added to the network. When a device is powered on, it sends this trap to its trap destination. If the trap destination is set correctly (i.e., to the IP address of destination NMS) the NMS can receive the trap and determine whether it needs to manage the device.
warmStart (1)	Indicates that the agent has reinitialized itself. None of the management variables will be reset.
linkDown (2)	Sent when an interface on a device goes down. The first

	variable binding identifies which interface went down.
linkUp (3)	Sent when an interface on a device comes back up. The first variable binding identifies which interface came back up.
authenticationFailure (4)	Indicates that someone has tried to query your agent with an incorrect community string; useful in determining if someone is trying to gain unauthorized access to one of your devices.
egpNeighborLoss (5)	Indicates that an *Exterior Gateway Protocol* (EGP) neighbor has gone down.
enterpriseSpecific (6)	Indicates that the trap is enterprise-specific. SNMP vendors and users define their own traps under the private-enterprise branch of the SMI object tree. To process this trap properly, the NMS has to decode the specific trap number that is part of the SNMP message.

RFC 1697 is the RDBMS MIB. One of traps defined by this MIB is *rdbmsOutOfSpace* :

```
rdbmsOutOfSpace TRAP-TYPE
   ENTERPRISE  rdbmsTraps
   VARIABLES   { rdbmsSrvInfoDiskOutOfSpaces }
   DESCRIPTION
      "An rdbmsOutOfSpace trap signifies that one of the database
       servers managed by this agent has been unable to allocate
       space for one of the databases managed by this agent. Care
       should be taken to avoid flooding the network with these traps."
   ::= 2
```

The enterprise is *rdbmsTraps* and the specific trap number is 2. This trap has one variable binding, *rdbmsSrvInfoDiskOutOfSpaces*. This variable is a scalar object. Its definition is:

```
rdbmsSrvInfoDiskOutOfSpaces OBJECT-TYPE
   SYNTAX  Counter
   ACCESS  read-only
   STATUS  mandatory
   DESCRIPTION
      "The total number of times the server has been unable to obtain
       disk space that it wanted, since server startup. This would be
       inspected by an agent on receipt of an rdbmsOutOfSpace trap."
   ::= { rdbmsSrvInfoEntry  9 }
```

The DESCRIPTION for this object indicates why the note about taking care to avoid flooding the network (in the DESCRIPTION text for the TRAP-TYPE) is so important. Every time the RDBMS is unable to allocate space for the database, the agent will send a trap. A busy (and full) database could end up sending this trap thousands of times a day.

Some commercial RDBMS vendors, such as Oracle, provide an SNMP agent with their database engines. Agents such as these typically have functionality above and beyond that found in the RDBMS MIB.

4.9.1 SNMP Notification

In an effort to standardize the PDU format of SNMPv1 traps (recall that SNMPv1 traps have a different PDU format from get and set), SNMPv2 defines a NOTIFICATION-TYPE. The PDU format for NOTIFICATION-TYPE is identical to that for *get* and *set*. RFC 2863 redefines the *linkDown* generic notification type like so:

```
linkDown NOTIFICATION-TYPE
   OBJECTS { ifIndex, ifAdminStatus, ifOperStatus }
   STATUS  current
   DESCRIPTION
          "A linkDown trap signifies that the SNMPv2 entity, acting in an agent
          role, has detected that the ifOperStatus object for one of its
          communication links left the down state and transitioned into some
          other state (but not into the notPresent state). This other state is
          indicated by the included value of ifOperStatus."
   ::= { snmpTraps 3 }
```

The list of bindings is called OBJECTS rather than VARIABLES, but little else has changed. The first object is the specific interface (*ifIndex*) that transitioned from the *linkDown* condition to some other condition. The OID for this trap is *1.3.6.1.6.3.1.1.5.3*, or *iso.org.dod.internet.snmpV2.snmpModules.snmpMIB.snmpMIBObjects.snmpTraps.linkDown*.

4.9.2 SNMP inform

Finally, SNMPv2 provides an inform mechanism, which allows for manager-to-manager communication. This operation can be useful when the need arises for more than one NMS in the network. When an inform is sent from one NMS to another, the receiver sends a response to the sender acknowledging receipt of the event. This behavior is similar to that of the get and set requests. Note that an SNMP inform can be used to send SNMPv2 traps to an NMS. If you use an inform for this purpose, the agent will be notified when the NMS receives the trap.

4.9.3 SNMP report

The report operation was defined in the draft version SNMPv2 but never implemented. It is now part of the SNMPv3 specification and is intended to allow SNMP engines to communicate with each other (mainly to report problems with processing SNMP messages).

4.10 Host Management

Managing hosts (i.e. computer systems) is an important part of network management. Though Host Resources MIB should seem to be a part of every host-based SNMP agent, but this isn't the case. Some SNMP agents implement this MIB, but many don't. A few agents go further and implement proprietary extensions based upon this MIB. This is mainly due to the fact that this MIB was intended to serve as a basic, watered-down framework for host management, designed mainly to foster wide deployment.

The Host Resources MIB defines the following seven groups:

```
host          OBJECT IDENTIFIER ::= { mib-2 25 }

hrSystem       OBJECT IDENTIFIER ::= { host 1 }
hrStorage      OBJECT IDENTIFIER ::= { host 2 }
hrDevice       OBJECT IDENTIFIER ::= { host 3 }
hrSWRun        OBJECT IDENTIFIER ::= { host 4 }
hrSWRunPerf    OBJECT IDENTIFIER ::= { host 5 }
hrSWInstalled  OBJECT IDENTIFIER ::= { host 6 }
```

The *host* OID is *1.3.6.1.2.1.25* (*iso.org.dod.internet.mgmt.mib-2.host*). The remaining six groups define various objects that provide information about the system.

The *hrSystem* (*1.3.6.1.2.1.25.1*) group defines objects that pertain to the system itself. These objects include uptime, system date, system users, and system processes.

The *hrDevice* (*1.3.6.1.2.1.25.3*) and *hrStorage* (*1.3.6.1.2.1.25.2*) groups define objects pertaining to filesystems and system storage, such as total system memory, disk utilization, and CPU nonidle percentage. They are particularly helpful, since they can be used to manage the disk partitions on your host. You can even use them to check for errors on a given disk device.

The *hrSWRun* (*1.3.6.1.2.1.25.4*), *hrSWRunPerf* (*1.3.6.1.2.1.25.5*), and *hrSWInstalled* (*1.3.6.1.2.1.25.6*) groups define objects that represent various aspects of software running or installed on the system. From these groups, you can determine what operating system is running on the host, as well as what programs the host is currently running. The *hrSWInstalled* group can be used to track which software packages are installed.

As you can see, the Host Resources MIB provides some necessary system-management objects that can be utilized by almost anyone who needs to manage critical systems.

References

[1] http://www.net-snmp.com, UCD-Davis Project

[2] http://www.ietf.org, TCP/IP Specification

[3] http://www.ietf.org, UDP Specification

Chapter Five

Design & Architecture

5.1 Design

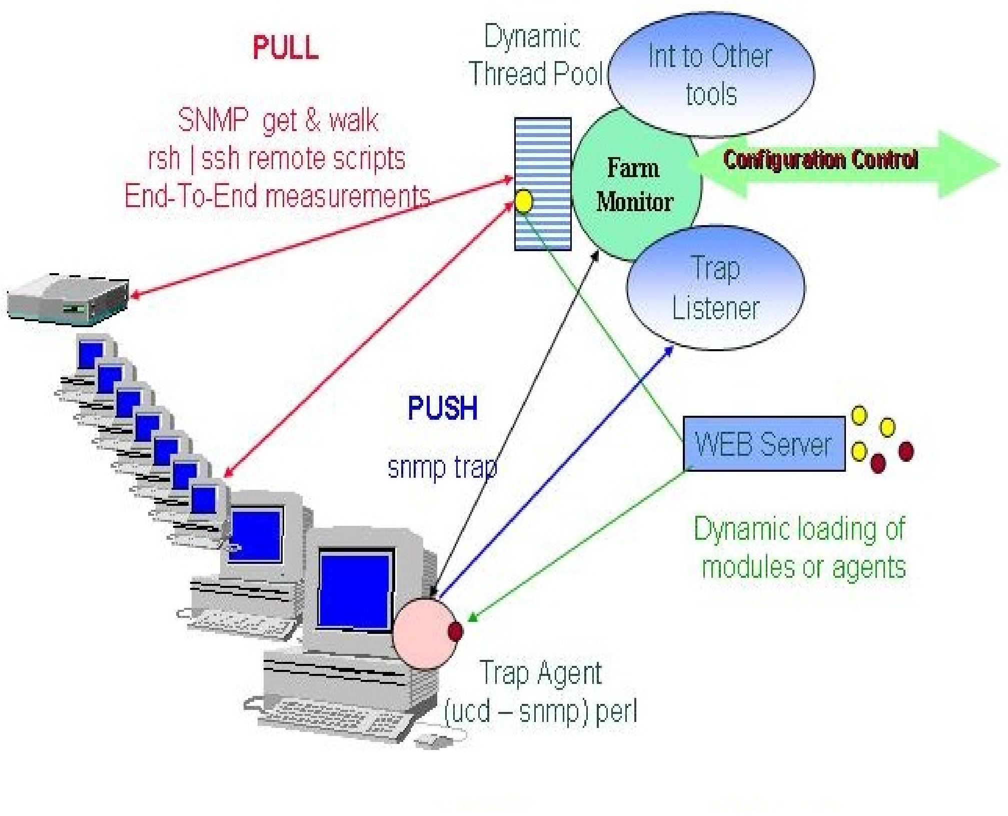


Figure 5-1: Basic Architecture diagram

5.2 Data Collection

The data collection part is based on a multithread engine that performs the procedures to get the requested values. A monitoring module performs a certain task (a SNMP request, runs a script or a performance program) and it can be dynamically loaded into the system and applied to a (set of) system(s) with a defined frequency. Dedicated modules to use parameters collected with other monitoring tools (e.g. Ganglia, MRTG) are controlled by the same multithread engine.

5.2 Monitoring Module

It is a dynamic loadable unit which executes a procedure (runs a script / program or performs SNMP request) to collect a set of parameters (monitored values) by properly parsing the output of the procedure. In general a monitoring unit is a simple class, which is using a certain procedure to obtain a set of parameters and report them in a simple, standard format.

5.3 Farm Monitoring Unit

This unit is responsible for the configuration and the monitoring of one or several farms. It can dynamically load any monitoring modules from a (set of) http servers or a distributed files system and use them to perform monitoring tasks on each node based on the configuration it receives from a RC Monitor unit. The dynamic pool of threads is used to run the specified monitoring modules on each node. This allows to run concurrently any number of modules and to efficiently use the resources. If performing a monitoring task fails or hangs due to I/O errors, the other tasks are not delayed or disrupted if such a multi-thread scheme is used. A dedicated control thread is used to stop the threads in case of I/O errors and to reschedule these tasks that have not been successfully done. A priority queue is used for the jobs that need to be executed periodically.

5.4 The Service System

Each farm unit may register as a JINI service and / or WSDL service. Clients or other services can get the system configuration and are notified when a change is done.
Access to monitor parameters is done using a predicate mechanism. Clients can subscribe with a predicate which may return historical data and / or perform a subscription for the matching values in the future. The monitoring predicates are based on regular expressions for string selection, including configuration parameters (e.g. system names. parameters), conditions for numerical values and time limits. In addition predicates may perform elementary functions like

MIN, MAX, average, integral. The predicate matching and the client notification are done in independent threads (one per client IP) under the control of the Data Cache unit. Measured values are currently stored into a relational DB using JDBC (InstantDB, MySQL, Postgres, Oralcle ...). The query procedures are adapted to the predicate mechanism for selecting historical data and at the same time to provide active listeners.

5.5 Filter Agents

More complex data processing can be done using Filter Agents. They are "active objects" which may be deployed by a client or an other service to perform a dedicated task using the data collected from a farm unit. It use a predicate to receive the data it needs and may send the computed values back a set of registered units. As an example, a maximum flow path algorithm can be done by such an agent. Agents may perform such tasks without being deployed to a certain service but in this case the Data Cache unit needs to send all the requested values to a remote site.

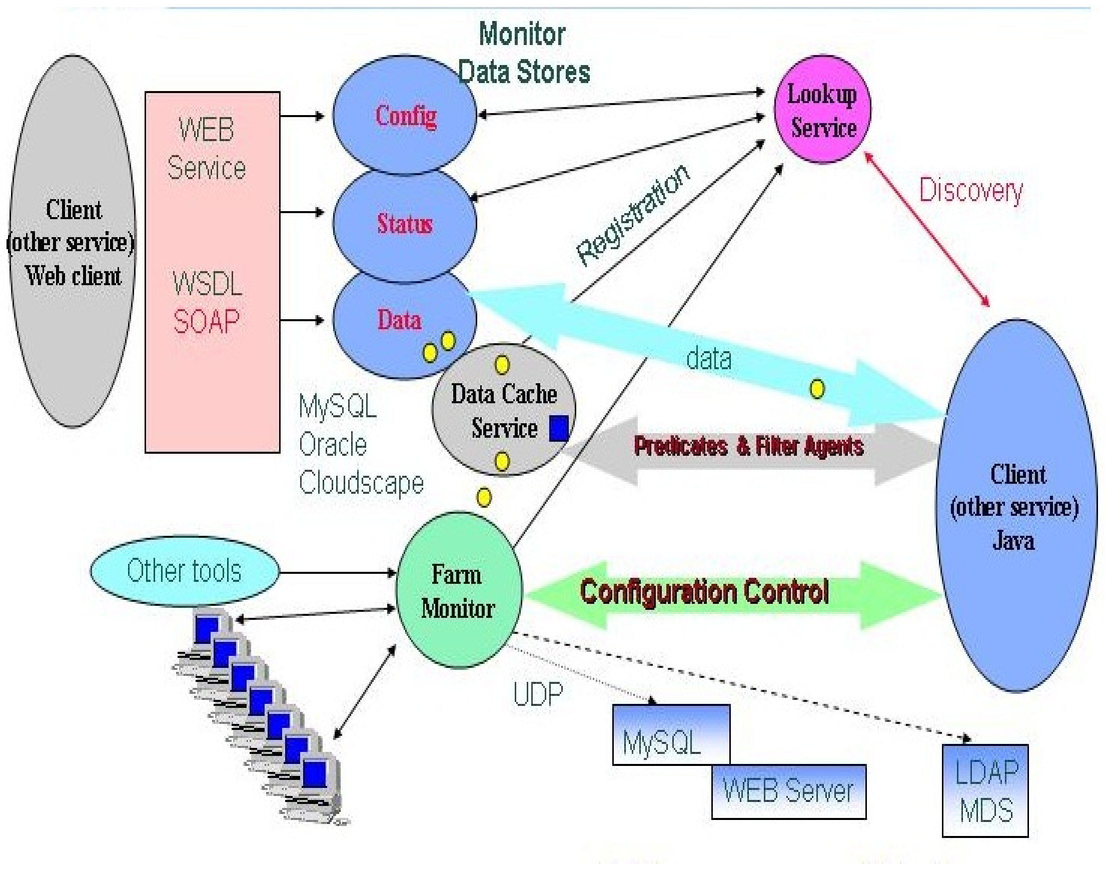


Figure 5-2 : Working Approach of the design

5.6 Auto Network Topology Discovery Module

A system to dynamically discover a large network with the heterogeneous topology of each node may it be router, manageable switch or computer system. This would allow management to its optimum locally. It should be able to discover the whole network so as to allow the dynamic loading of monitoring modules easier. In large and constantly evolving networks, it is difficult to determine how the network is actually laid out. Yet this information is invaluable for network management, simulation, and server placement. Traditional topology discovery algorithms are based on SNMP,

5.7 Conclusion

Monitoring modules were completed by the end of this phase save a few for which research continued in order to retrieve the information to deal with the heterogeneity problems of different operating systems. Auto Network Topology Discovery Module required a continuous and thorough literature study therefore it kept carried on.

Chapter Six

Auto Network Topology Discovery

6.1 Introduction

Network topology is a representation of the interconnection between directly connected peers in a network. In a *physical* network topology, peers are ports on devices connected by a physical transmission link. A physical topology corresponds to many *logical* topologies, each at a different level of abstraction. For example, at the IP level, peers are hosts or routers one IP hop from each other, and at the workgroup level, the peers are workgroups connected by a logical link. In this paper, by network topology we refer exclusively to the logical IP topology, ignoring hubs and bridges, and link-level details such as FDDI token rotation times, ATM or Frame-relay links, and Ethernet segment lengths. At this level, a peer corresponds to one or more IP addresses, and a link corresponds to a channel with specific delay, capacity, and loss characteristics.

Network topology constantly changes as nodes and links join a network, personnel move offices, and network capacity is increased to deal with added traffic. Keeping track of network topology manually is a frustrating and often impossible job. Yet, accurate topology information is necessary for:

- *Simulation*: In order to simulate real networks, the topology of the network must be first obtained.
- *Network Management*: Network topology information is useful in deciding whether to add new routers and to figure out whether current hardware is configured correctly. It also allows network managers to find bottlenecks and failures in the network.
- *Siting:* A network map helps users determine where they are in the network so they can decide where to site servers, and which ISP to join to minimize latency and maximize available bandwidth.

- *Topology-aware algorithms*: Topology information enables a new class of protocols and algorithms that exploit knowledge of topology to improve performance. Examples include topology-sensitive policy and QoS routing, and group communication algorithms with topology-aware process group selection.

Thus, there is a considerable need for automatic discovery of network topology. Currently, the only effective way to do so is by exploiting SNMP (Simple Network Management Protocol). However, there are many situations where SNMP cannot be used. SNMP is not implemented in older machines, and in newer machines, SNMP may be turned off or have restricted access. In a growing heterogeneous network, where decisions about the network and access are decentralized, it is naive to assume that SNMP has been installed, and is accessible, on every node in the network. We think that this situation is the norm, rather than the exception.

Design

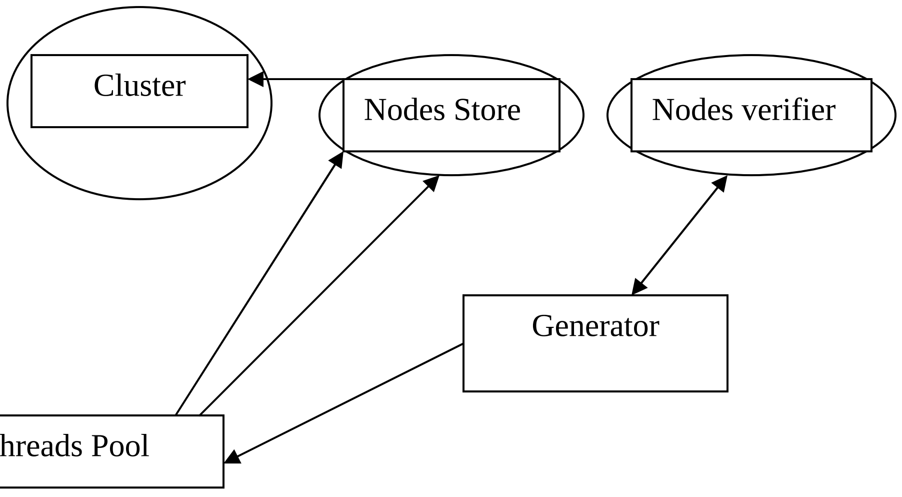


Figure 6-1: Design of Auto Network Topology Module

Algorithm

Since this Monitoring Service depends upon the SNMP protocol for monitoring of the resources therefore the mainly algorithm depends upon this protocol. Steps are as follows

Since this Monitoring Service depends upon the SNMP protocol for monitoring of the resources therefore the mainly algorithm depends upon this protocol. Steps are as follows

a. Host to the application is queried for it’s gateway plus the active connections it has, IP Addresses of all are retrieved. These IPs are sent to Node Store to be picked up by the Nodes Generator.

b. When Nodes Generator picks up the IPs, it removes them from Nodes Store and hand the IPs over to Thread Scheduler.

c. Thread scheduler generate the Nodes and place a copy of their all local IPs in Nodes verifier to be checked later if the Node has already been generated and the node object is stored in Cluster through Node Store.

d. Whenever a Node is generated, its remote connections and gateway is also submitted to the Node Store for Nodes to be generated queue, to be generated later.

6.4 Packages

6.4.1 Monitor.modules.ActiveCluster.SnmpCluster

This package contains the hierarchical structure of a cluster and it’s nodes with the monitoring modules, to find out different information regarding the active nodes with active remote connections. About nine different types of monitoring modules have been compiled within one single Class SnmpInfo.

6.4.2 Monitor.monitor

This package contains the snmpwalk class file that actually interacts with the joesnmp library on behalf of the monitoring module for the required information. Two different types of walking functions have been provided that could retrieve either a scalar or a tabular mib object, from the destination agent. Another file that has been added to this package is AppConfig which allows setting of the properties in order to dynamically configure the program.

6.4.3 Monitor.util

It contains the thread pooling classes for the scheduling of node generation in a huge environment. Moreover it contains the priority queue for the jobs posted earlier to be executed in queue unless priority specifies otherwise.

6.4.4 Monitor.NodeStore

Since this Monitoring Service depends upon the SNMP protocol for monitoring of the resources therefore the mainly algorithm depends upon this protocol. Steps are as follows It contains the classes that rearranges the Nodes according to the network classes of its' IPs. Different Levels have been included with every class that shall allow multiple nodes to load the monitoring modules at a time.

6.4.5 Gui

It contains classes to retrieve from the Node Store and Nodes Generator. It may be provided with the arguments like IP or host name to start from plus the SNMP community string. Else it uses the defaults that is the localhost as the starting node and 'public' as the SNMP community string.

6.4.6 test

It has been provided to give user a touch of this module running without actively using the module over their network. A test file is provided to pick the objects from.

6.5 Figures

Figures of this module have been provided at the Appendix

Chapter 7
Conclusion and Future work

7.1 Project Summary

This Monitoring Service will provide real support for Higher Level Loosely coupled services in co-ordination and it will be the mainly used tool in the CMS production activities (visualization, debugging, optimize resource utilization) and at the same time will act as flexible high-level dynamic service to be used by any other grid related applications or distributed network services, which require such data.

Currently this service is being run in many universities including NUST and has brought good name to the university by providing its resources as a FARM to LHC data GRID.

Main addition here this year was the monitoring modules and Auto-network topology discovery module which in itself has been a full fledge project. This will help discovering different nodes in a huge collection of networks with the topology, thereby providing an ease to the administrators and Monitoring Service.

7.2 Achievements

The achievements so far are as under:

a. Service is deployed in more than six universities around the world. It is capable of monitoring any number of farms (networks) that may be dynamically added to it.
b. Auto Network Topology Discovery Module has been developed and shall be integrated soon into the project
c. Different Monitoring Modules have been developed and integrated.
d. A very sophisticated multi threading mechanism, with dynamic pool of running threads has been implemented in the Discovery Module.

7.3 Future Research

Progress is already very encouraging as the system that has been developing as a prototype has been installed many where thereby giving a chance of improvement in all aspects, by different feedbacks from different places. Following may be included in the future research

a. Integrating the GRID desirable features in the Service.
b. Integrating Fault Tolerance Measures
c. Making interface for the mobile agents.
d. Making Interface for JXTA [1] Communities and clients.

7.4 Conclusion

The project has been carried out due to NUST-CALTECH collaboration and is supposed to be mainly used in CERN LHC Data Grid. Many developers have been since years developing and suggesting different designs in NUST and CALTECH both. Our achievement remained that we added new features to the service like enhancing GUI, adding Monitoring Modules and above all the Auto Network Topology Discovery Module which in itself was a big enough project.

References

[1] http://www.jxta.org, JXTA Specification

Appendix "A"

Understanding MIB

The following example is a stripped-down version of MIB-II (anything preceded by -- is a comment):

```
RFC1213-MIB DEFINITIONS ::= BEGIN

        IMPORTS
              mgmt, NetworkAddress, IpAddress, Counter, Gauge,
              TimeTicks
                    FROM RFC1155-SMI
              OBJECT-TYPE
                    FROM RFC 1212;

        mib-2      OBJECT IDENTIFIER ::= { mgmt 1 }

-- groups in MIB-II

        system       OBJECT IDENTIFIER ::= { mib-2 1 }
        interfaces   OBJECT IDENTIFIER ::= { mib-2 2 }
        at           OBJECT IDENTIFIER ::= { mib-2 3 }
        ip           OBJECT IDENTIFIER ::= { mib-2 4 }
        icmp         OBJECT IDENTIFIER ::= { mib-2 5 }
        tcp          OBJECT IDENTIFIER ::= { mib-2 6 }
        udp          OBJECT IDENTIFIER ::= { mib-2 7 }
        egp          OBJECT IDENTIFIER ::= { mib-2 8 }
        transmission OBJECT IDENTIFIER ::= { mib-2 10 }
        snmp         OBJECT IDENTIFIER ::= { mib-2 11 }

        -- the Interfaces table

        -- The Interfaces table contains information on the entity's
        -- interfaces. Each interface is thought of as being
        -- attached to a 'subnetwork.' Note that this term should
        -- not be confused with 'subnet,' which refers to an
        -- addressing-partitioning scheme used in the Internet
        -- suite of protocols.

        ifTable OBJECT-TYPE
           SYNTAX  SEQUENCE OF IfEntry
           ACCESS  not-accessible
           STATUS  mandatory
           DESCRIPTION
        "A list of interface entries. The number of entries is
         given by the value of ifNumber."
    ::= { interfaces 2 }

ifEntry OBJECT-TYPE
    SYNTAX  IfEntry
    ACCESS  not-accessible
    STATUS  mandatory
    DESCRIPTION
        "An interface entry containing objects at the subnetwork
         layer and below for a particular interface."
    INDEX   { ifIndex }
    ::= { ifTable 1 }

IfEntry ::=
    SEQUENCE {
        ifIndex
            INTEGER,
        ifDescr
            DisplayString,
        ifType
            INTEGER,
        ifMtu
            INTEGER,
        ifSpeed
            Gauge,
        ifPhysAddress
            PhysAddress,
        ifAdminStatus
            INTEGER,
        ifOperStatus
            INTEGER,
        ifLastChange
            TimeTicks,
        ifInOctets
            Counter,
        ifInUcastPkts
            Counter,
        ifInNUcastPkts
            Counter,
        ifInDiscards
            Counter,
        ifInErrors
            Counter,
        ifInUnknownProtos
            Counter,
            ifOutOctets
               Counter,
            ifOutUcastPkts
               Counter,
            ifOutNUcastPkts
               Counter,
            ifOutDiscards
               Counter,
            ifOutErrors
               Counter,
            ifOutQLen
               Gauge,
            ifSpecific
               OBJECT IDENTIFIER
         }

      ifIndex OBJECT-TYPE
         SYNTAX  INTEGER
         ACCESS  read-only
         STATUS  mandatory
         DESCRIPTION
            "A unique value for each interface. Its value ranges
             between 1 and the value of ifNumber. The value for each
             each interface must remain constant at least from one
             reinitialization of the entity's network-management
             system to the next reinitialization."

         ::= { ifEntry 1 }

      ifDescr OBJECT-TYPE
         SYNTAX  DisplayString (SIZE (0..255))
         ACCESS  read-only
         STATUS  mandatory
         DESCRIPTION
            "A textual string containing information about the
             interface. This string should include the name of
             the manufacturer, the product name, and the version
             of the hardware interface."
         ::= { ifEntry 2 }

END
```

The first line of this file defines the name of the MIB, in this case RFC1213-MIB. (RFC 1213 is the RFC that defines MIB-II; many of the MIBs we refer to are defined by RFCs). The format of this definition is always the same. The IMPORTS section of the MIB is sometimes referred to as the *linkage* section. It allows you to import datatypes and OIDs from other MIB files using the IMPORTS clause. This MIB imports the following items from RFC1155-SMI (RFC 1155 defines SMIv1, which we discussed earlier in this chapter):

- mgmt
- NetworkAddress
- IpAddress
- Counter
- Gauge
- TimeTicks

It also imports OBJECT-TYPE from RFC 1212, the *Concise MIB Definition*, which defines how MIB files are written. Each group of items imported using the IMPORTS clause uses a FROM clause to define the MIB file from which the objects are taken.

The OIDs that will be used throughout the remainder of the MIB follow the linkage section. This group of lines sets up the top level of the *mib-2* subtree. *mib-2* is defined as *mgmt* followed by *.1*. We saw earlier that *mgmt* was equivalent to *1.3.6.1.2*. Therefore, *mib-2* is equivalent to *1.3.6.1.2.1*. Likewise, the *interfaces* group under *mib-2* is defined as { mib-2 2 }, or *1.3.6.1.2.1.2*.

After the OIDs are defined, we get to the actual object definitions. Every object definition has the following format:

```
<name> OBJECT-TYPE
   SYNTAX <datatype>
   ACCESS <either read-only, read-write, write-only, or not-accessible>
   STATUS <either mandatory, optional, or obsolete>
   DESCRIPTION
      "Textual description describing this particular managed object."
   ::= { <Unique OID that defines this object> }
```

The first managed object in our subset of the MIB-II definition is *ifTable*, which represents a table of network interfaces on a managed device (note that object names are defined using mixed case, with the first letter in lowercase). Here is its definition using ASN.1 notation:

```
ifTable OBJECT-TYPE
   SYNTAX  SEQUENCE OF IfEntry
   ACCESS  not-accessible
   STATUS  mandatory
   DESCRIPTION
      "A list of interface entries. The number of entries is given by
       the value of ifNumber."
   ::= { interfaces 2 }
```

The SYNTAX of *ifTable* is SEQUENCE OF IfEntry. This means that *ifTable* is a table containing the columns defined in *IfEntry*. The object is not-accessible, which means that there is no way to query an agent for this object's value. Its status is mandatory, which means an agent must implement this object in order to comply with the MIB-II specification. The DESCRIPTION describes what exactly this object is. The unique OID is *1.3.6.1.2.1.2.2*, or *iso.org.dod.internet.mgmt.interfaces.2*.

Let's now look at the SEQUENCE definition from the MIB file earlier in this section, which is used with the SEQUENCE OF type in the *ifTable* definition:

```
IfEntry ::=
   SEQUENCE {
      ifIndex
         INTEGER,
      ifDescr
         DisplayString,
      ifType
         INTEGER,
      ifMtu
         INTEGER,
      .
      .
      .
      ifSpecific
         OBJECT IDENTIFIER
   }
```

Note that the name of the sequence (*IfEntry*) is mixed-case, but the first letter is capitalized, unlike the object definition for *ifTable*. This is how a sequence name is defined. A sequence is simply a list of columnar objects and their SMI datatypes, which defines a conceptual table. In this case, we expect to find variables defined by *ifIndex*, *ifDescr*, *ifType*, etc. This table can contain any number of rows; it's up to the agent to manage the rows that reside in the table. It is possible for an NMS to add rows to a table. This operation is covered later, in the section "The set Operation."

Now that we have *IfEntry* to specify what we'll find in any row of the table, we can look back to the definition of *ifEntry* (the actual rows of the table) itself:

```
ifEntry OBJECT-TYPE
   SYNTAX  IfEntry
   ACCESS  not-accessible
   STATUS  mandatory
   DESCRIPTION
      "An interface entry containing objects at the subnetwork layer
      and below for a particular interface."
   INDEX   { ifIndex }
   ::= { ifTable 1 }
```

ifEntry defines a particular row in the *ifTable*. Its definition is almost identical to that of *ifTable*, except we have introduced a new clause, INDEX. The index is a unique key used to define a single row in the *ifTable*. It's up to the agent to make sure the index is unique within the context of the table. If a router has six interfaces, *ifTable* will have six rows in it. *ifEntry*'s OID is *1.3.6.1.2.1.2.2.1*, or *iso.org.dod.internet.mgmt.interfaces.ifTable.ifEntry*. The index for *ifEntry* is *ifIndex*, which is defined as:

```
ifIndex OBJECT-TYPE
   SYNTAX  INTEGER
   ACCESS  read-only
   STATUS  mandatory
   DESCRIPTION
      "A unique value for each interface. Its value ranges between
       1 and the value of ifNumber. The value for each interface
       must remain constant at least from one reinitialization of the
       entity's network-management system to the next reinitialization."
     ::= { ifEntry 1 }
```

The *ifIndex* object is read-only, which means we can see its value, but we cannot change it. The final object our MIB defines is *ifDescr*, which is a textual description for the interface represented by that particular row in the *ifTable*. Our MIB example ends with the END clause, which marks the end of the MIB. In the actual MIB-II files, each object listed in the *IfEntry* sequence has its own object definition. In this version of the MIB we list only two of them, in the interest of conserving space.

Appendix B

Textual Conventions for Creation of Managed Objects

SMIv2 also introduces new textual conventions that allow managed objects to be created in more abstract ways. RFC 2579 defines the textual conventions used by SNMPv2, which are listed in table below

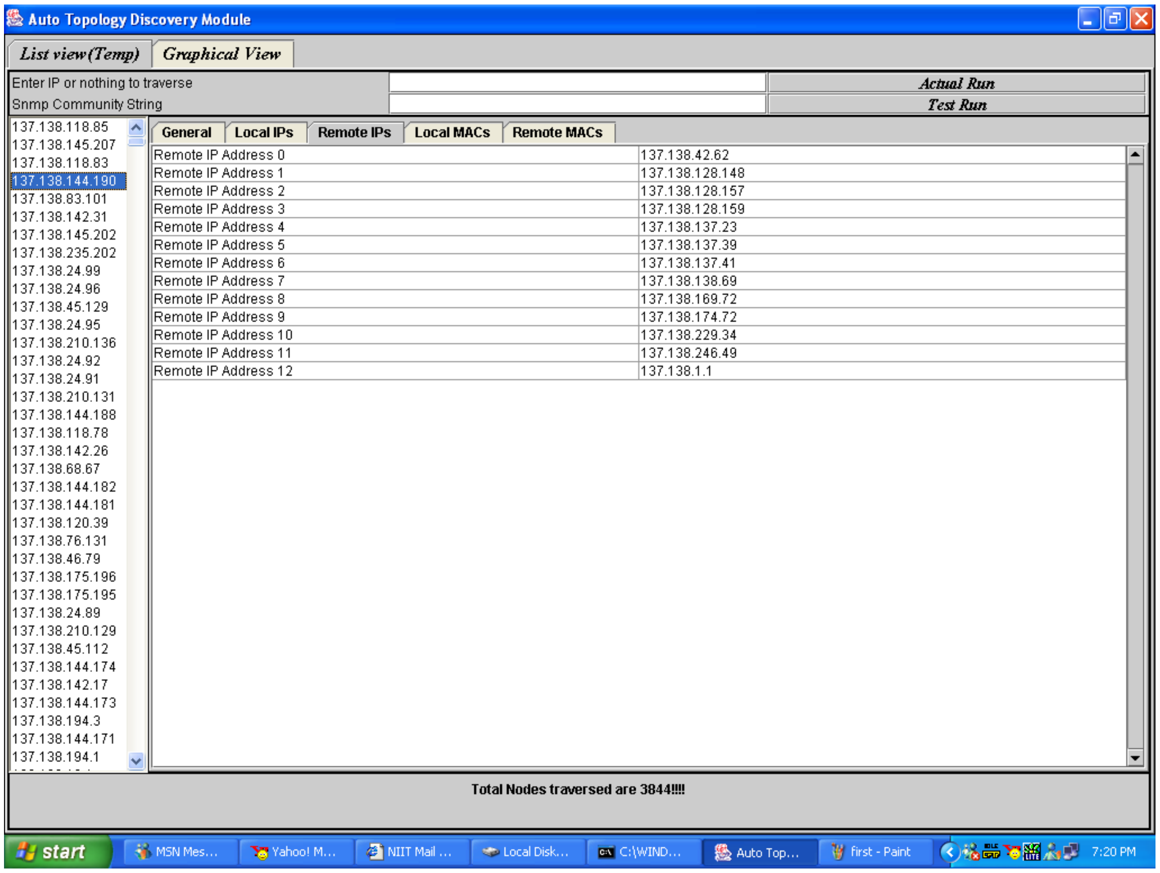
Textual Conventions for SMIv2

Textual Convention	**Description**
DisplayString	A string of NVT ASCII characters. A DisplayString can be no more than 255 characters in length.
PhysAddress	A media- or physical-level address, represented as an OCTET STRING.
MacAddress	Defines the media-access address for IEEE 802 (the standard for local area networks) in canonical[5] order. (In everyday language, this means the Ethernet address.) This address is represented as six octets.
TruthValue	Defines both true and false Boolean values.
TestAndIncr	Used to keep two management stations from modifying the same managed object at the same time.
AutonomousType	An OID used to define a subtree with additional MIB-related definitions.
VariablePointer	A pointer to a particular object instance, such as the *ifDescr* for interface *3*. In this case, the VariablePointer would be the OID *ifDescr.3*.
RowPointer	A pointer to a row in a table. For example, *ifIndex.3* points to the third row in the *ifTable*.
RowStatus	Used to manage the creation and deletion of rows in a table, since SNMP has no way of doing this via the protocol itself. RowStatus can keep track of the state of a row in a table, as well as receive commands for creation and deletion of rows. This textual convention is designed to promote table integrity when more than one manager is updating rows. The following enumerated types define the commands and state variables: active(1), notInService(2), notReady(3), createAndGo(4), createAndWait(5), and destroy(6).
TimeStamp	Measures the amount of time elapsed between the device's system uptime and some event or occurrence.
TimeInterval	Measures a period of time in hundredths of a second.

	TimeInterval can take any integer value from 0-2147483647.
DateAndTime	An OCTET STRING used to represent date-and-time information.
StorageType	Defines the type of memory an agent uses. The possible values are other(1), volatile(2), nonVolatile(3), permanent(4), and readOnly(5).
TDomain	Denotes a kind of transport service.
TAddress	Denotes the transport service address. TAddress is defined to be from 1-255 octets in length.

Appendix “C”

Auto Network Topology Discovery



Enter IP or nothing to traverse

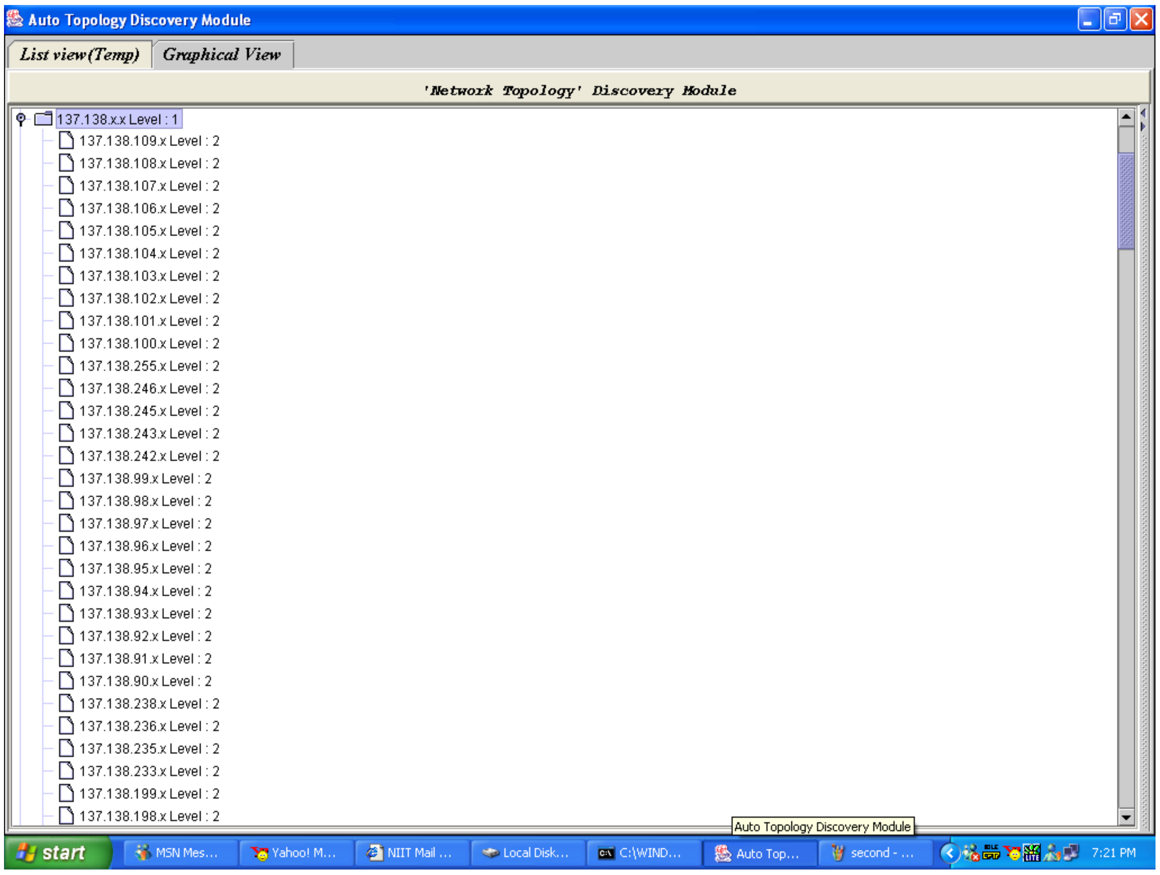
Snmp Community String

```
137.138.118.85
137.138.145.207
137.138.118.83
137.138.144.190
137.138.83.101
137.138.142.31
137.138.145.202
137.138.235.202
137.138.24.99
137.138.24.96
137.138.45.129
137.138.24.95
137.138.210.136
137.138.24.92
137.138.24.91
137.138.210.131
137.138.144.188
137.138.118.78
137.138.142.26
137.138.68.67
137.138.144.182
137.138.144.181
137.138.120.39
137.138.76.131
137.138.46.79
137.138.175.196
137.138.175.195
137.138.24.89
137.138.210.129
137.138.45.112
137.138.144.174
137.138.142.17
137.138.144.173
137.138.194.3
137.138.144.171
137.138.194.1
```

Remote IP Address 0	137.138.42.62
Remote IP Address 1	137.138.128.148
Remote IP Address 2	137.138.128.157
Remote IP Address 3	137.138.128.159
Remote IP Address 4	137.138.137.23
Remote IP Address 5	137.138.137.39
Remote IP Address 6	137.138.137.41
Remote IP Address 7	137.138.138.69
Remote IP Address 8	137.138.169.72
Remote IP Address 9	137.138.174.72
Remote IP Address 10	137.138.229.34
Remote IP Address 11	137.138.246.49
Remote IP Address 12	137.138.1.1

Total Nodes traversed are 3844!!!!

Fig A : Nodes and their Related Information after Detection of the Complete Network


Auto Topology Discovery Module
List view(Temp)
Graphical View
'Network Topology' Discovery Module

```
137.138.x.x Level : 1
    137.138.109.x Level : 2
    137.138.108.x Level : 2
    137.138.107.x Level : 2
    137.138.106.x Level : 2
    137.138.105.x Level : 2
    137.138.104.x Level : 2
    137.138.103.x Level : 2
    137.138.102.x Level : 2
    137.138.101.x Level : 2
    137.138.100.x Level : 2
    137.138.255.x Level : 2
    137.138.246.x Level : 2
    137.138.245.x Level : 2
    137.138.243.x Level : 2
    137.138.242.x Level : 2
    137.138.99.x Level : 2
    137.138.98.x Level : 2
    137.138.97.x Level : 2
    137.138.96.x Level : 2
    137.138.95.x Level : 2
    137.138.94.x Level : 2
    137.138.93.x Level : 2
    137.138.92.x Level : 2
    137.138.91.x Level : 2
    137.138.90.x Level : 2
    137.138.238.x Level : 2
    137.138.236.x Level : 2
    137.138.235.x Level : 2
    137.138.233.x Level : 2
    137.138.199.x Level : 2
    137.138.198.x Level : 2
```

Fig B: Tree View of different Levels of Network Classes

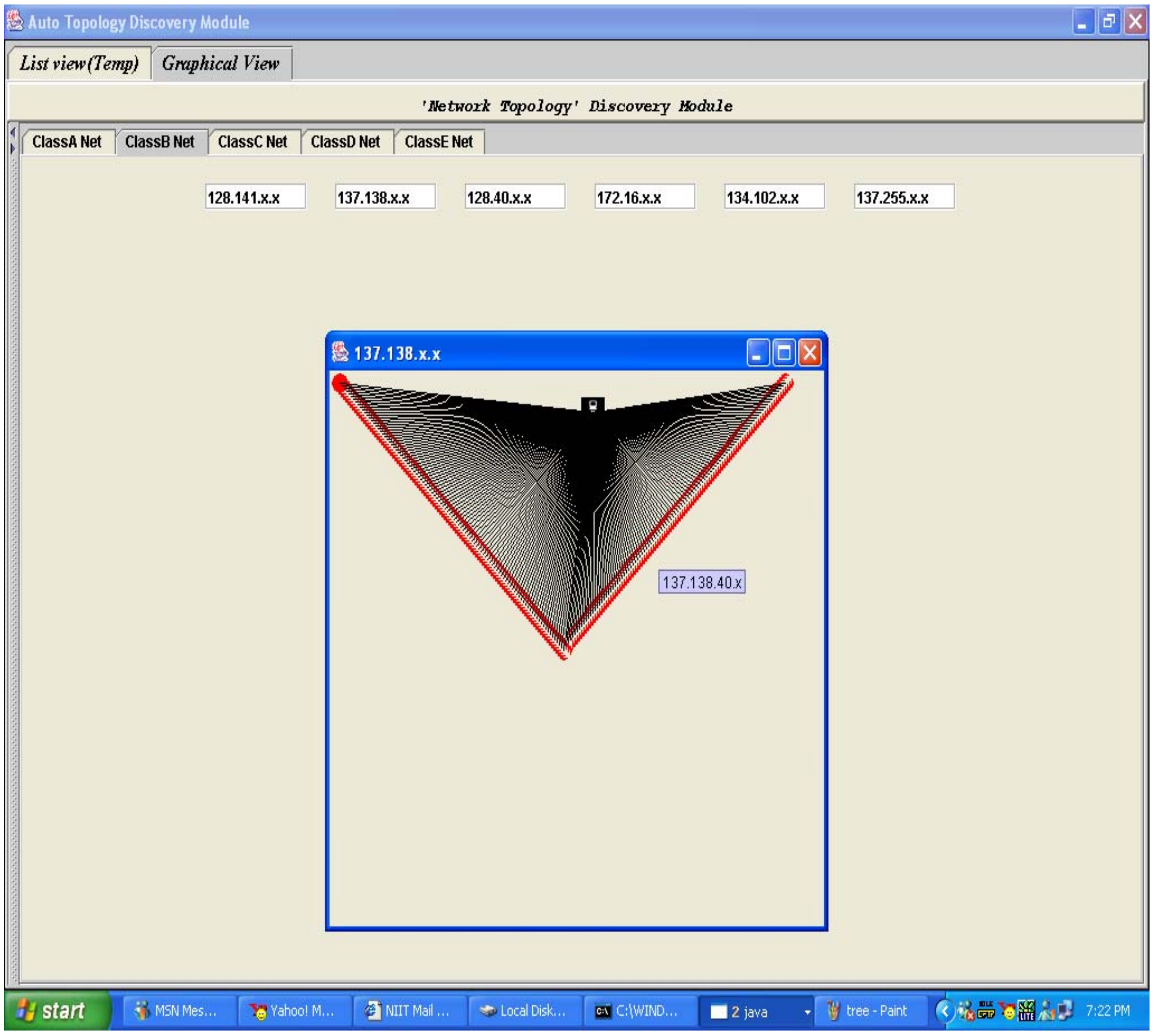


Fig C : Nodes of Class B are connected to their Gateway

Appendix "D"

MonaLisa User Guide

version 0.4 24 Dec 2002

This Monitoring Service has a different name in CALTECH and CERN and is known as MonaLisa (Monitoring Agents using Large Integrated Services Architecture). It is entirely written in java. The distribution is packed in several self contain jar files. Practically it is required to have only java installed on the system hosting the MonaLisa Service (NOT on the monitored nodes!). The way to configure and use the service is described in this guide. MonaLisa also allows to dynamically add new monitoring modules, Filters or Agents.

Java

For running a MonaLisa service you need to have the java runtime environment (jse 2 1.4 or higher)
installed on one   system that will run the Monitoring Service for an entire farm.

To set the environment to run java may look like this :

```
JAVA_HOME=$HOME/JAVA/jdk
export JAVA_HOME
export PATH=$JAVA_HOME/bin:$PATH
```

SNMPD

MonaLisa has dedicated modules to collect values provided by snmp demons. It is recommended that the snmpd demon is installed and configured. The snmp demon should run on the all the systems you would like to monitor, including switches or routers.

You can test the values provided by the snmp demon (based on how it is configured) by using

```
snmpwalk  [-p port_no]  system_name community .1
```

If snmp is not used, it is possible to collect monitoring values provided by tools like Ganglia or any shell, perl scripts .
(please see Ganglia Interface)

DataBase system

MonaLisa comes with an embedded SQL Data Base (InstantDB) which is used by default . If you are happy with this option nothing has to be done. If you would like to use an other DB system, you need to have the JDBC driver for it and to create initially the tables used by MonaLisa. We tested the system with MySQL and Postgres. The MonaLisa distribution provides the drivers and the configuration scripts for the two DataBase systems. We also provide for convenience an archive which has all the scripts to install and configure MySQL to be used as storage mechanism for monitoring values. To install and use MySQL it is not required root access. In $MonaLisa_HOME/Examples there are simple scripts (in SimpleEx1_MySql/ and SimpleEx1_PostgreSQL/) for using MonaLisa with MySQL or PostgreSQL. The JDBC drivers can be found in $MonaLisa_HOME/DBdrivers/.

Authentication & Security

The Administration of the service can be done from a GUI and is done via a SSL connection. This requires the user to provide X.509 certificate to be imported into the key trust store of the service. Once the user's certificate becomes a trusted certificate for the service it allows the Administrator user to change the configuration from a GUI. This is required only if the user want to change the configuration.

Configuration

A simple configuration file is used to define the configuration to be monitored. This file is used at the startup of the services to define the clusters, nodes and the monitor modules to be used. The configuration, the monitored parameters and the repetition time may be changed via de Administrative GUI.

Setup the configuration files for your site

- Go inside the "MonaLisa" directory and create a directory for your site. You may copy the configuration files from
  one of the available site directory (e.g.: under RCs : CITCMS, UCSD, CernLXBARCH...).

* Edit the configuration file (your_site.conf) to reflect the environment you want to monitor.

* You may add a myIcon.gif file with an icon of your organization which will be used to

Start a Monitoring Service

In the examples mentioned earlier there are scripts to start the monitoring applications.
The script run_sitename can be used to start a standalone service to monitor the configuration you created.
 The script jrun_sitename starts the service as the previous one and register it with a set of Lookup Discovery Services. In this case this service can be seen by anyone in the federation.
In these scripts it is possible to set or change default parameters used by the service. A ReadMeOptions.txt file list all the parameters and the default values used. Alternatively these values may be provided in an option file.

This scripts use MonaLisa_HOME environment variable which we should set to where the MonaLisa directory is located.
In MonaLisa directory a sent_env script may be used for this or simply :

export MonaLisa_HOME=/your_path/MonaLisa

Once you provide the optional parameters in this scripts you can start the service :
./run_sitename
./jrun_sitename if you want the service to be registered as a global service.

Start an Administration GUI

In order to connect to a MonaLisa service as an administrator your X.509 certificate must first be imported into the MonaLisa service
keystore as a trusted entry. The MonaLisa keystore is located under the SSecurity directory. A script to import a certificate into the keystore
can be found in the same directory. We also provide scripts to create a new keystore.

To start a MonaLisa Admin GUI :
cd Admin
./run system_name
where system_name is the name of the system running a MonaLisa service

Start a GUI for a service :

This allows oneone to see and generate plots of the collected values .
cd Clients/Gui

./run_GUI system_name
where system_name is the name of the system running a MonaLisa service

Start a Global GUI

This program allows to discover all the registered centers with any set of Lookup Discovery services
(started with jrun command) and allows any user to see real-time global values as well any measurement for each component
in the system and its history .
cd Clients/Gui/
./jGlobal

You may want to change the list of Lookup Discovery services to be used (simply edit the jGlobal script)

Web Service Client WSDL/SOAP (java)

MonaLisa provides also a Web Service interface. It allows any client to connect and receive selected values. A predicate mechanism can be used . In Clients/Web you may find simple examples.

Web Service Client WSDL/SOAP (perl)

Examples in using perl to access the MonaLisa Web Service interface are provided in Clients/perl

Writing into MDS or any other programs or database systems

The MonaLisa framework allows to dynamically load additional modules for writing (or processing) the collected values.
in usr_code/MDS is an example of writing the received values into MDS. This is done using a unix pipe to communicate between the
dynamically loadable java module and the script performing the update into the LDAP server.
Please also read the ReadMe file from this directory.
An other simple example which simply print all the values on sysout can be found on usr_code/SimpleWriter
An other example to write the values into UDP sockets is in usr_code/UDPWriter

Writing new Monitoring Modules

New Monitoring modules can be easily developed. These modules may use SNMP requests or can simply run any script (locally or on a remote system) to collected the requested values. The mechanism to run these modules under independent threads, to perform the interaction with the operating system or to control an snmp session are inherited from a basic monitoring class. The user basically should only provide the mechanism to collect the values. to parse the output and to generate a result object. It is also required to provide the names of the parameters this module is collecting.

Examples to generate new modules are in user_code/Modules/

Writing new Filters

Filters allow to dynamically create any new type of derived value from the collected values. Es an example it allow to evaluate the
integrated traffic over last n minutes, or the number of nodes for which the load is less than x. filters may also send an email to a list or SMS messages when predefined complex condition occur. These filters are executed in independent threads and allow any client to register for its output. They may be used to help application to react on certain conditions occur, or to help in presenting global values for large computing facilities.