END HOST MONITORING AGENT

By

Aziz Allauddin
Savera Tanwir
Waleed Mansoor

SUBMITTED TO THE FACULTY OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY,
RAWALPINDI
IN PARTIAL FULFILLMENT FOR THE REQUIREMENTS OF A B.E
DEGREE IN
COMPUTER SOFTWARE ENGINEERING

MAY 2004

No portion of the work presented in this dissertation has been submitted in support of another award or qualification either at this institution or elsewhere

Abstract

Large scale distributed systems such as computational and data grids require pervasive end-to-end resource monitoring for effective problem diagnoses, performance analysis, performance tuning and job scheduling. Gathering monitoring information can be a tedious task in itself. The complexity in retrieving such an information originates from the fact that not only we require knowledge of end systems but also the intervening network segments, over which we may not have any control or administrative privileges, like the Internet. The End host monitoring Agent (EMA) is a java based, dynamically configurable monitoring tool, which enables the user for easy data acquisition from end-hosts. The information comprises of static and dynamic performance data such as the system, memory, CPU, disk and network parameters. It has built-in mechanisms for regularly scheduled measurements and reporting of results to any data repository. EMA is a lightweight tool and has a scalable, multilayered architecture. In this report we have discussed the need for such an application, its architecture, design and applications in detail. In the end results and some future tasks are enlisted.

ACKNOWLEDGEMENTS

The EMA development team is thankful to Almighty Allah for the successful completion of the project.

We thank our parents for their excellent support not only during the course of this project, but also throughout our lives for without them, all this would have been impossible.

We are also thankful to our project supervisors; Dr. Arshad Ali, Director NIIT, Dr. Iosif Legrand, Senior Software Engineer at Caltech, and Ahmad R. Shahid, Instructor at NUST, for their guidance, support and instructive supervision throughout the project.

Acknowledgements are also due to HoD CS Department at MCS, Lt. Col. Raja Iqbal, Lt. Col Nadeem, MCS faculty members and administration, who in spite of their busy schedule provided guidance and support, not only during this work but also throughout the course of the degree.

PREFACE

The development of high performance networks, such as Internet2's Abilene network, has enabled researchers to achieve high performance but only under certain conditions for some advanced network applications. The users find a gap between the potential of such high performing network infrastructure and their own experience. It is difficult to locate a problem since little or no information is externally available about an underlying network. To overcome these issues, an end-to-end view of the Internet, which involves the network path as well as the hosts, the protocols and the applications, is required. The End host Monitoring Agent (EMA) was developed as an effort to address these issues. This report aims to make the reader understand the need and importance of resource monitoring. It gives an analysis of existing tools and techniques and provides a new infrastructure for host and network performance monitoring in the distributed environment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CERN - European Organization for Nuclear Research

CVS – Concurrent Versioning System

DLL – Dynamic Link Library

GigaPoP – Giga bit Point of Presence

GUI – Graphical User Interface

EMA – End host Monitoring Agent

E2Epi- End-to-End Performance Initiative

E2EpiPE - End-to-End Performance Initiative Performance Environment

JNI – Java Native Interface

JNLP – Java Native Language Protocol

JWS – Java Web Start

LAN – Local Area Network

MonALISA – Monitoring Agents using a Large Integrated Service Architecture

NIC- Network Interface Card

OS – Operating System

PMP – Performance Monitoring Points

PDH – Performance Data Helper

QoS – Quality of Service

RTT – Round Trip Time

SNMP – Simple Network Management Protocol

UCAID - University Corporation for Advanced Internet Development

WMI - Windows Management Infrastructure

WAN – Wide Area Network

1

INTRODUCTION

1.1 Background

There has been tremendous advancements in the field of computer networks/data communications, over the past few years. Data transfer rates have improved by a factor of 1 million in just 25 years [1]. Today we have long haul, high speed networks, capable of transferring data at the rate of Giga bits per second. However, despite the advancements in network technology, users still have to contend with less than optimal performance. This stands true for the high speed bandwidth links used for scientific research as well as ordinary TCP communication. Performance and quality of service is vital for active research; especially where huge amounts of data are transferred over the networks.

When a user experiences a performance bottleneck with an application that employs long-haul, high-speed Internet connections, it can become very difficult to locate the cause of the problem. Even if the problem is located, finding the qualified person to fix it, is even harder. This is primarily due to the decentralized nature of the Internet, where each network operator can run his/her network with very little coordination with other network operators. While this approach has created great flexibility and independence in building the Internet, and is one of the main reasons the Internet has seen exponential growth in so short span a time, it has also hampered the provision of any service/s that needs cooperation across operating domains.

Currently, little or no operational information is externally available about a network domain, which makes it particularly difficult to locate problems. This necessitates the development of tools that could help in locating and troubleshooting such problems. A

number of such tools have been developed that provide valuable information regarding various different aspects of network performance; but a single tool normally focuses on just one aspect of performance measurement and/or requires a relatively advanced technical knowledge about networks and protocols in order to interpret the results. This makes it infeasible for users from other domains to use such tools. This also leads to the "Wizard Gap" [2].

To overcome the above mentioned hurdles, and to enable high network performance, the need was felt to develop an infrastructure that fulfills the requirements:

1. A mechanism for gathering necessary information from the hosts and network segments.
2. Availability of centralized information storage/analysis engine, capable of problem detection.
3. Problem diagnoses/reporting mechanism.
4. Development of an easy to use user interface, which lets an end user, with little knowledge of the network, understand the capabilities of his network path.

Such an infrastructure will provide the end users with a simplified but efficient way of solving performance issues.

Internet2 End-to-End Performance Initiative Performance Environment System (E2E piPEs), is one such framework, that is still under development, that will be able to

indicate performance capabilities and locate performance problems along the path between two computers connected through the UCAID Abilene network, participating campuses, regional networks, and gigaPoPs. [3]. It aims to fulfill all the above mentioned objectives. Such an infrastructure will significantly improve the likelihood of advanced Internet applications operating at peak performance and thereby increasing the productivity of researchers.

The key to developing such an infrastructure lies in the use of existing tools and techniques in a way that integrates them to create more comprehensive and easy to use tools. Moreover, accurate and effective diagnostics of network problems not only requires information about the networks, but issues related to end-hosts involved in the communication are also important. This is due to the fact that it is the host that provides the user with an interface to the network and to maintain throughput levels, the host system must be able to move data from the application buffers, through the kernel, and onto the network interface buffers at a speed faster than that of the network interface. The hosts represent the logical dividing point between the "network" and user.[3] Quite a few performance problems are related to host configuration. This includes problems related to TCP (the most widely used protocol), operating system, NIC, firmware, application etc. Therefore, it is essential that information about hosts should be gathered and its performance monitored over a period of time.

1.2 Scope

The concept of de-centralized monitoring is a rarity in majority of the monitoring tools currently available. Mostly, nodes are monitored via SNMP (Simple Network Management Protocol). This puts heavy load on the servers and causes a lot of network congestion. To remove these bottlenecks, a monitoring tool was designed keeping in view the de-centralized monitoring architecture, where each node was responsible for reporting the parameters to central server. However, such a tool should not be heavy on the resources rather it should consume negligible resources of the end-host. Once developed, it can be deployed across LANs, WANs, different types of Grids and also at the same time serve as the primary monitoring client end of Internet2.

1.3 Aim

The aim of this report is to highlight the features offered by End host Monitoring Agent : an application for gathering and representing host and network information and to bring the efforts and course of development of EMA into limelight so that others might use it and further advance it.

2

LITERATURE REVIEW

2.1 What is Grid?

The term “Grid” was coined in the mid 1990s to denote a proposed distributed computing infrastructure for advanced science and engineering research [4]. Considerable progress has since been made on the construction of such an infrastructure. Increasingly, computing is concerned with collaboration, data sharing, and other new modes of interaction that involve distributed resources. The result is an increased focus on the interconnection of systems both within and across enterprises, whether in the form of intelligent networks, switching devices, caching services, appliance servers, storage systems, or storage area network management systems. These enhancements in services generate new requirements for distributed application development and deployment. Today, applications and middleware are typically developed for a specific platform (e.g., Windows NT, a flavor of Unix, a mainframe, J2EE, Microsoft .NET) that provides a hosting environment for running applications.

But in spite of this diversity, the continuing decentralization and distribution of software, hardware, and human resources makes it essential that desired qualities of service (QoS) are achieved on resources assembled dynamically from enterprise systems, service provider systems, and customer systems. QoS can be measured in terms of common security semantics, distributed workflow and resource management performance, coordinated fail-over, problem determination services, or other metrics. New abstractions and concepts are needed that allow applications to access and share resources and services across distributed, wide area networks.

Such problems have for quiet sometime posed concerns central importance to the developers of distributed systems for large-scale scientific research. Work within this community has led to the development of *Grid technologies* [4, 5], which address precisely these problems and which are witnessing widespread and successful adoption for scientific and engineering computing.

2.2 Benefits of Grid

Grid concepts and technologies were first developed to enable resource sharing within far-flung scientific institutions [6, 7, 8]. Applications included collaborative visualization of large scientific datasets (pooling of expertise), distributed computing for computationally expensive data analyses (pooling of compute power and storage), and coupling of scientific instruments with remote computers and archives (increasing functionality as well as availability) [4]. Similar applications are expected to become important in commercial settings, initially for scientific and technical computing applications and then for commercial distributed computing applications, including enterprise application integration and business to business (B2B) partner collaboration over the Internet. Just as the World Wide Web began as a technology for scientific collaboration and was adopted for e-business, for the future seems equally bright for Grid technologies and computing.

2.3 Monitoring the Grid

An essential part of managing a global Data Grid is a monitoring system that is capable of monitoring and tracking of the many site facilities, networks, and the many tasks in progress, in real time. The monitoring information gathered is also essential for

developing the required higher level services, and components of the Grid system that provide decision support, and eventually some degree of automated decisions, to help maintain and optimize workflow through the Grid.

Grid platforms depend on monitoring and provision of information services to support the discovery and monitoring of the distributed resources for various tasks. For example, a user may want to determine the best platform on which to run an application, a client program may want to collect a stream of data to help steer an application, a system administrator may want to be notified when changes in the system load occur or free disk space is available or a user having problems with the network may want to know the bottlenecks to fix them. For all the above mentioned problems, a performance monitoring infrastructure was needed.

2.4 Internet 2's End-to-End Performance Initiative

Internet2 is a consortium led by over 200 universities working in partnership with industry and government to develop and deploy advanced network applications and technologies, accelerating the creation of tomorrow's Internet. The aim of the Internet2 project [9] is to recreate the original partnership among academia, industry and government towards the three-fold goal of creating a leading edge network capability for the national research community, enabling revolutionary Internet applications and ensuring the rapid transfer of new network services and applications to the broader Internet community.

Internet2 allows for "grid" computing, which enables distributed global problem solving.

Internet2 provides immediate access to resources and experiences otherwise unavailable over the "commodity" Internet.

Recent experience within the Internet2 community has shown that even with high bandwidth connections campus researchers do not often see concurrent performance. Research into this apparent performance disparity reveals bottlenecks and problems at various points along the end-to-end network path. A network engineer or an applications specialist would find bottlenecks and problems at a variety of points along the end-to-end network path between what the researcher sees on the screen and the resource at the other end of the connection. The main objective of the Internet2 End-to-End Performance Initiative [10] was to identify common problems, identify tools and techniques to detect and solve these problems, and to define and implement an operational environment where a researcher could expect to use the full capability of the network on a regular basis. Advances in the operational deployment and use of active and passive measurements and in the robustness of end-hosts and applications are expected as a result of this Initiative, which includes participation from campuses, network operators, network researchers, and vendors of network and computing equipment.

The goal of this project is to aggregate tools and integrate resulting data to allow existing measurement techniques to be used over an end-to-end path to determine the performance of various segments that make up that path. The project will also use the work of others to create a repository of performance data that would be available for use by all people. This data repository should help reduce the number of active tests (possibly redundant in

nature) on the network by providing access to recent tests that meet the requirements of the requestor. The project will also develop an easy to use tool interface that lets an end user, with little knowledge of the network, understand the capabilities of the network path. If the performance is less than expected, the system will assist the end user in determining which link is creating the bottleneck and who to contact to get it fixed. The system will also keep a record of the tests so that they may be passed on to the contact experts, showing the reason that a particular segment of the path is suspected of mis-behavior.

2.5 E2EpiPE System Architecture

2.5.1 Engineering Assumptions and Goals

The E2E piPE system was built on the following network engineering assumptions [11]:

1. The general problem of inadequate end-to-end performance has a wide variety of causes and cannot be solved by a single tool and/or a single viewpoint on the network..
2. End-to-end performance problems, almost by definition, cross boundaries between multiple autonomous administrative domains.
3. End-to-end performance problems can occur at points in between two intended end points. Thus the aim is to identify where in the intended path lies the problem
4. End-to-end performance problems are often experienced by end users who are not highly trained network engineers and who may not be intimately familiar with the implementation details of the advanced network application they are employing.
5. While many good tools useful in solving end-to-end performance problems exist, understanding how to use them correctly, understanding how to interpret their

results, and knowing who to contact to fix a particular problem is something beyond the expertise of the most end users.

6. Solutions aimed at solving end-to-end performance problems should not significantly degrade performance and/or increase security risks to the network.

The design goals for the project are:

1. Achieve modular, open source design, allowing other researchers to contribute to the ongoing evolution of the product.
2. Achieve decentralized control with no single points of failure.
3. Enable voluntary participation of multiple administrative domains, as end-to-end performance problems inherently span multiple administrative domains.
4. Create a hierarchical system of authentication and authorization that prevents denial-of-service attacks from either undermining or being launched from the system.
5. Make collected network data available to researchers for uses beyond those envisioned by this project.
6. Make intelligent reuse of existing public domain software, as limited resources preclude reinventing the wheel.
7. Collaborate closely with other ongoing projects by other groups, so there is no unnecessary duplication of effort.

2.6 Performance Elements

A system-level view of the Internet encompasses host platforms, which include their hardware, operating system (OS) and application software and other network path

characteristics of the intervening segments connecting the host platforms.

End-to-end performance involves host system characteristics such as memory, I/O, bandwidth, and CPU speed; the OS; and the application implementation. To maintain throughput levels, the host system must be able to move data from the application buffers, through the kernel, and onto the network interface buffers at a speed faster than that of the network interface. Apart from that the network segments play a major role in determining the performance. Also there can be configuration limitations and network protocol issues.

Broadly we can identify three major problem areas: network configuration, network protocols, and applications.

"Starting from the host is a logical first step which in this regard and it has the potential to make the biggest impact". [12] The hosts represent the logical dividing point between the "network" and the "user" therefore we must enable "data acquisition" from the hosts involved. Many problems are related to host configs, TCP/IP stacks, OS version, NICs, firmware, application design, etc.

The system should be able to dynamically download a platform independent data gathering application which can extract relevant host information that basically is divided into three categories: stable, dynamic and network information.

Also the system is able to conduct network path testing between two nodes. The test could be a defined set of measurements, Ping (reachability/RTT), One-way loss (each

direction) and Iperf (bandwidth EACH way, measured simultaneously)

2.7 Need for EMA

In order to carry out the above mentioned tests, a system is presented that has the capability of gathering system level information from the end-host, supporting the two most widely used operating systems i.e. Windows and Linux. The series of tests can be run by a Java application as java provides platform independence. The application can be dynamically downloaded from the web servers. Implementation of the Java Network Launching Protocol relieves the user from the drudgery of installation and configuration of the application. In addition to that, the application provides the user with the facility to carry out bandwidth measurement tests through an easy to use interface to already existing tools. Dynamic host information is recorded at both ends. Provision for reporting the monitored values to a remote server for possible analysis also exists. These Logged events could be further analyzed by central servers with access to current network details. Users could be referred to the most likely problem domain with current contact information provided by the central server.

3

INCEPTION

3.1 Project Vision

Dr. Iosif C. Legrand is a Professor at California Institute of Technology (Caltech). As Caltech is part of Internet 2 collaboration, Dr. Iosif is an important member of Internet 2 organization and is responsible for development of monitoring infrastructure for Internet 2. For this purpose, currently he is working at CERN on the LHC Grid project. There is an ongoing collaboration between NUST and CERN. Dr. Arshad Ali is supervising number of projects that are being developed for LHC Grid. Dr. Arshad and Dr. Iosif after mutual round of discussions, proposed that a project be carried out to develop a monitoring software for Internet 2 which follows E2Epi monitoring architecture to some extent. After some format selection procedures, and some informal discussions, we were selected as group members with Dr. Arshad and Dr. Iosif as supervisiors.

The central idea of the project was to develop a monitoring system, which gathered performance data of various components of the network and present the user with a high level as well as detailed view of the Network performance.

3.2 The Nature of the System

The major portion of the system consisted of a subsystem, which gathered real time performance parameters of the local machine. This included a number of parameters like CPU Usage, number of threads etc. The network performance was measured by carrying out different type of tests between various nodes.

3.3 General Requirements

A list of general requirements of the above mentioned system is given below:

1. The system should measure performance of the hosts as well as the network path between the hosts.
2. The system should be platform-independent; it should work exactly the same way on both Linux and Windows.
3. The system should be dynamically configurable; it should not require any complex installation procedures and instructions.
4. The system should support remote monitoring.
5. The system should be scalable so that it can accommodate new tools and techniques without affecting the existing modules.
6. The system should be highly integratable to allow its integration into large Grid Monitoring Infrastructure.
7. The application should have a comprehensive and easy to use Graphical User Interface.

3.4 Functional Requirements

The general requirements gave an overview of the functionalities that the system should encompass. The next step was to divide the system into subsystems that would help realize the use case model. Splitting the functionality at this early stage helped to further unfold the problem to be solved. The modules identified are enumerated below:-

1. Host Performance monitoring modules
    a. Windows Performance monitoring Modules

    b. Linux Performance Monitoring Modules

2. Network Performance Monitoring Modules
    a. Module for Abing
    b. Module for Iperf
3. Data management module
4. Information reporting module
5. Remote monitoring module
6. Server Module for information receipt
7. Performance Data Storage Module
8. Client Graphical User Interface
9. Administrator Graphical User Interface

3.5 An Overview of Measurement Infrastructure

The measurement infrastructure collects and provides all the basic measurement data used to determine performance. The infrastructure consists of all the measurement devices that collect the basic measurements and stores them for use in analysis.

3.5.1. Measurement Attributes

Measurement attributes are the data that is associated with a particular measurement. This includes not only the particular measurement value but also information that indicates how, where and when the measurement was taken.

3.5.1.1. Measurement Type

This attribute is the main aspect of measurement collected. There are two types of

measurements Network Measurements and Host/Application Measurements.

The Network Measurements consist of Percentage Packet Loss, Bandwidth and Round Trip Time. While the Host/OS/Application measurements consist of Memory Usage, CPU Load, Number of processes and threads, Available and used memory, Disk Usage, Pages in and out, Disk space and the Network interface traffic.

3.5.2. Measurement Method

There are different methods of obtaining measurements. This section will look at some of those methods and discuss some pros and cons of each method.

3.5.2.1. Active Measurement

"Active Measurement" sends sampling data along a path to determine the network performance. It may be desirable to develop reference traffic profiles for different types of applications that can be used as benchmarks. However active measurement also consumes the resources that it measures. To prevent all network bandwidth being consumed by active measurements, they need to be limited in use.

3.5.2.2. Passive Measurement

"Passive measurements" are taken by observing the existing traffic. Passive measurements can easily be used to determine gross performance characteristics, for example measuring packets delivered through an interface. Passive measurements become more difficult with increased detail. It is also more difficult to select the appropriate data for passive measurement as the speed increases. However passive

measurements have the advantage that they do not affect the network performance.

3.5.3. Information Exchange Format

The Information Exchange Format is the specification of how the Measurement Attributes will be presented when transferred to a device requesting the information. The Format could also influence the way the data is stored in Data Repositories. Although the Measurement Attributes will use the format for representation, the format should be designed to be flexible enough to accommodate all future Measurement Attributes.

3.5.4. Data Repository

A data repository is a collection of measurement data from an area of the measurement infrastructure. It will contain information from all methods of measurement, active, passive, etc. A data repository may also do some aggregation or analysis on the data and store the results for later retrieval by the clients.

4

ELABORATION

4.1 Design and Development Decisions

After carrying out research and thorough analysis of the problem domain, following design and development decisions were taken.

4.1.1 Platform Independence

The Internet2 comprises of heterogeneous networks. Moreover the hosts forming the networks are also diverse in nature. Different operating systems plus different architecture/OS combinations are present. The monitoring system should be able to deal with this heterogeneity. It should be platform independent and give a uniform view to the user, no matter what the underlying platform may be.

Java was thus naturally selected as the language of development, as it is supported on many platforms. However different OS' have different mechanisms for extraction of performance related information. Java does not have a uniform API for performance monitoring. Therefore it was decided that different Performance Monitoring Modules should be developed for different Operating Systems. Appropriate modules can be loaded at runtime and will give the user same interface regardless of underlying OS.

Different OS' make available different parameters and not all are available on every platform (for instance Load is not available on Windows Platform). Thus it became one of the parts of this project to implement such missing parameters.

Windows and Linux were chosen as the two Platforms for which development was

undertaken. As these two platforms comprise more than 95 % of the systems in Internet2. All versions of Windows which are NT based (like Win 2000 and Win XP) were supported. All major flavors of Linux like RedHat , Slackware, Debian are also supported.

4.1.2 Dynamic Downloading

One of the major requirements of the system was that it should be as much user friendly as possible. Therefore the application was made dynamically downloadable. Java Webstart Technology was selected for this purpose, which uses Java Network Launching Protocol. (For further discussion of Java Webstart see appendix 3). This relieves the user of the hassle of installation and upgrading. Moreover it makes it easy for the developers to distribute the latest versions of the software by just placing it on the web servers.

4.1.3 Integration with existing tools

For network performance measurement part, many tools are available which take one measurement or the other. Examples of these tools are Web100, Iperf, Abing and Ping. Rather than re-inventing the wheel, integration of tools became the main aim of the project.

4.1.4 Highly Modular

Design of the application was kept highly modular so as to cope with changing requirements. The design has been kept so much modular that if a new performance monitoring module is added, only a single line of code needs to be added in one of the

classes and the rest of the modules (like the Control Layer, the GUI, and the database layer) adopt accordingly.

4.2 Processes, Tools and Techniques used

Table 4-1: Processes, Tools and Techniques

Development process	Rational Unified Process
Analysis & Design	Object Oriented Analysis and Design
Programming Language	Java and Visual C++
GUI Designing	Java API
Web access	Java Web Start
Network Communication	UDP and TCP
Main Development IDE	EditPlus, KWrite
Linux Desktop	KDE

Table 4-2: Additional Support Systems

Versioning System	CVS (Concurrent Versioning System)
Video Conferencing System	VRVS (Virtual Room Video Conferencing System)
Database Server	mySQL
Web Server	Microsoft IIS, Apache
API Documentation	Java doc 1.4.2, MSDN
FTP	Linux wuftp System
Build tool	Ant

4.3 Choice of database Server

The monitoring system maintains a central repository of all the monitoring information for a certain period of time. For this a database is needed. The three main requirements the design specifies for the database are that it should be open source and free, it should be really fast and lightweight and lastly it should be supported by JDBC. After comparing and analyzing many open source databases, mySQL was selected. It fulfills all the above requirements and its response time is faster than some other commercial databases like the Oracle [13].

5

CONSTRUCTION

The iterative development approach was chosen, as it has been identified to be the most favorable methodology for a successful development cycle by RUP. Thus designing and redesigning of the system was carried out, to improve working of the system and to rectify flaws, if any.

5.1 Design and Implementation Directives

The system comprises of a Client application gathering host characteristics and reporting them to the repository, a server to receive the values that are integrated with the data repository and an Administrator Interface for remote monitoring and data analysis.

5.2 Architecture

EMA has client-server architecture. The EMA clients run on each host and report performance statistics to a central Server. The server has its own data repository to store the received parameters. Apart from examining the reported host characteristics, the administrator of the central server can also remotely monitor the network path between any two EMA nodes. The over all scenario is depicted in the figure 5.1. Both the clients and the server have separate graphical user interfaces.

Keeping in view the future requirements for retrieving new parameters, and integrating new tools, the overall application architecture has been kept sufficiently modular. This modular approach is also according to the guidelines of E2E performance initiative. Furthermore the architecture is layered to separate the parameter categorizing and presentation logic from parameter extracting logic.

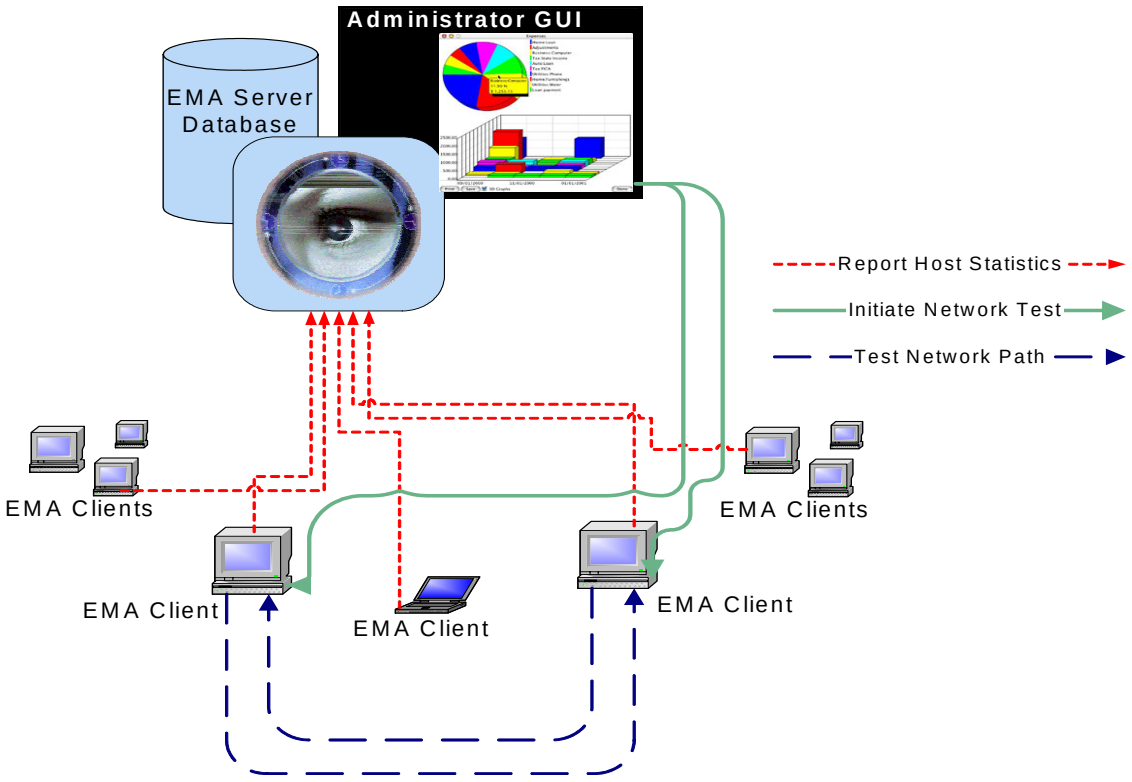

Figure 5.1– Architecture

5.2.1 EMA Client Architecture

EMA Client monitors three types of information: static, dynamic and network. They are further categorized into three classes i.e. stable, dynamic and network information.

The Stable Information is least likely to change during a single session. It includes the parameters, IP Address, MAC Address, System, Architecture, Kernel, User, Java Vendor and Java Version.

In contrast to that the Dynamic Information covers system parameters that are changed at a rapid pace and reflect the most recent state of the system as a whole. Dynamic information can be divided into sub-categories as CPU, System, Memory, Disk and Network Interface.

The CPU category covers different areas of CPU usage. The information displayed comprises of the overall CPU being used, the percentage of CPU being used by system and user processes and the percentage of processor(s) being spent in idle state.

The System category parameters display the current state of the system in terms of number of processes and the number of threads. It also provides a description of average load over the last 1, 5 and 15 minutes.

Memory describes the amount of memory usage, used memory, free memory, page in/sec and page out/sec.

Parameters in the Disk category show the rate of disk I/O along with used and available disk space statistics.

Network Interface comprises of information pertaining to traffic I/O for all network interfaces present on the host.

Network Information includes the information of the network path between the host systems. This information is collected on demand and consists of Bandwidth which is the bandwidth to and from another host, RTT that gives the round trip time and Packet Loss which represents the packet loss in the path between two hosts on the network.

Parameters like those discussed above can be used to diagnose problems concerning the host system characteristics/configuration. Some of these parameters such as “CPU Usage” can be directly obtained from the underlying operating system, while others like “System Load” have to be derived from a number of other parameters. Furthermore, new categories and parameters can be easily added. Moreover EMA also integrates some

available tools for obtaining statistics about the network.

EMA client architecture is divided into three layers. This layered architecture is shown in figure 5.2.The first/lowest layer comprises of modules that interact with host operating system and collect raw parameters. Different operating systems provide different mechanisms for retrieving system information. This requires some platform specific functionality to be added. To cater for this requirement, different sets of these modules have been developed for each operating system. Currently EMA supports Windows and Linux platforms. The application identifies the underlying operating system at runtime, and appropriate modules are activated.

Second layer gathers the dynamic system information from the first layer. It validates and arranges them into categories. It also presents a simple interface to the third/GUI layer, which displays the information for the user. GUI design has been kept highly adaptive, so that if new parameter gathering module is added at the lower layer at some later stage, the GUI needs not be redesigned, rather it adjusts itself accordingly. Also EMA client can run without the GUI and can act as a base layer to higher level performance tools. Thus overall design of the application is highly scalable and facilitates easy integration with the new tools.

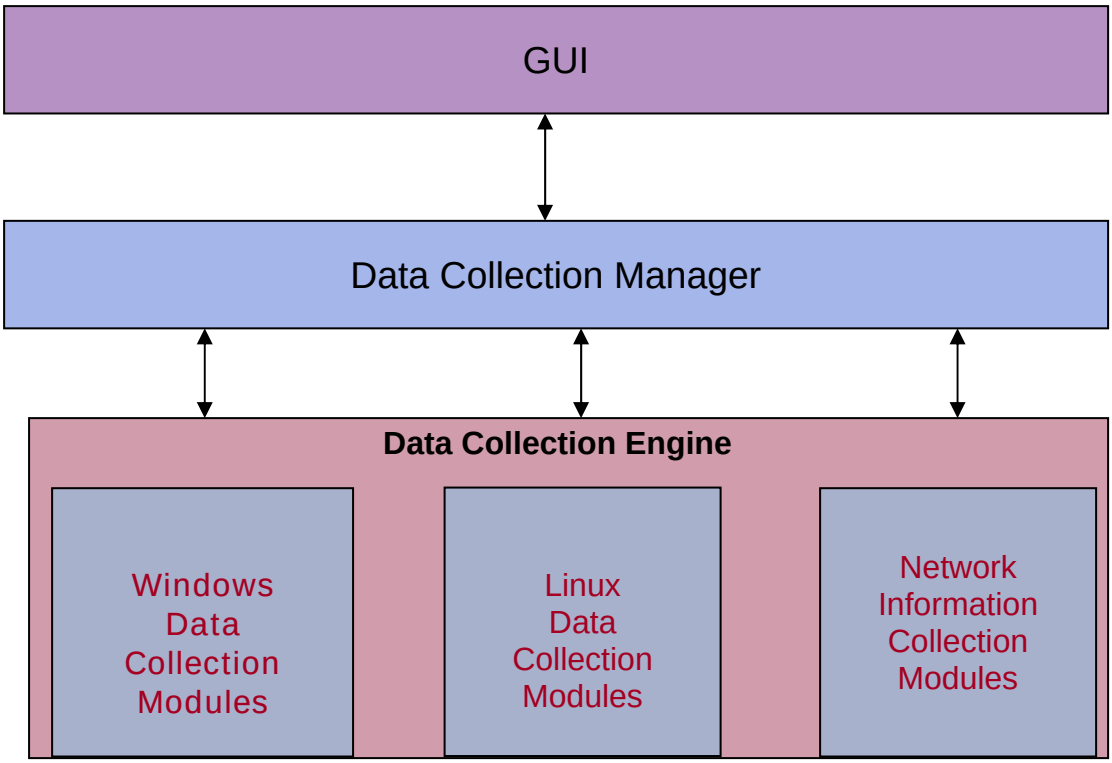

Figure 5.2 - EMA Client Architecture

5.2.1.1 DATA COLLECTION ENGINE

The main purpose of the data collection engine is to gather static and dynamic host information from the operating system kernel, and the network information from the tools like Abing and Iperf. Abing is a tool using the packet pairs dispersion technique to estimate the available Bandwidth/bitrate (unused capacity) for a path in the network and Iperf is a tool to measure maximum TCP bandwidth, allowing the tuning of various parameters and UDP characteristics. Iperf reports bandwidth, delay jitter, datagram loss.

The data collection engine can be broadly divided into three categories: Windows, Linux and Network monitoring modules. A description of each is given below.

5.2.1.1.1 Windows Modules

Performance information is extracted from the Windows operating system using a 2-layer abstraction. This is shown in figure 5.3. The upper layer is implemented in java; it obtains performance data from the lower layer modules and passes it onto the data collection manager. The lower layer is implemented in Visual C++ by using the Windows Native API. Performance Data Helper (PDH) library has been used which is a native library for manipulating performance information for a Windows based system. The Performance Data Helper (PDH) is a companion library to the native performance-monitoring features of the Windows NT operating system. It is built on top of the standard performance-monitoring features of Windows NT and doesn't really add any new functionality to native performance monitoring.

The performance data that the Windows NT operating system provides contains information for a variable number of object types, instances per object, and counters per object type. The counters are used to measure various aspects of performance. For example, the Process object includes the Handle Count counter to measure the number of handles open by the process. An instance is a unique copy of a particular object type, though not all object types support multiple instances. For example, the System object has no instances since there is only one System. On the other hand, the Process object supports multiple instances because Windows NT supports multiple processes.

PDH functions obtain performance data provided by WMI through providers that use performance extension DLLs or the high-performance data provider object. [14]

By default, the operating system obtains performance data for system resources using the registry. When we use performance tools to access registry functions for performance

data, the system collects the data from the appropriate system object managers, such as the Memory Manager, the input/output (I/O) subsystem, and so forth. [15]

As an option, Windows 2000 supports collecting data using the Windows Management Infrastructure (WMI) interface. In addition to several of the system performance counter DLLs, the operating system installs managed object files (MOFs) for data collection using WMI instead of the registry. These files reside in System32\Wbem\Mof. The Windows Management service must be running on the monitoring and monitored computer (if different) in order to obtain data using WMI.

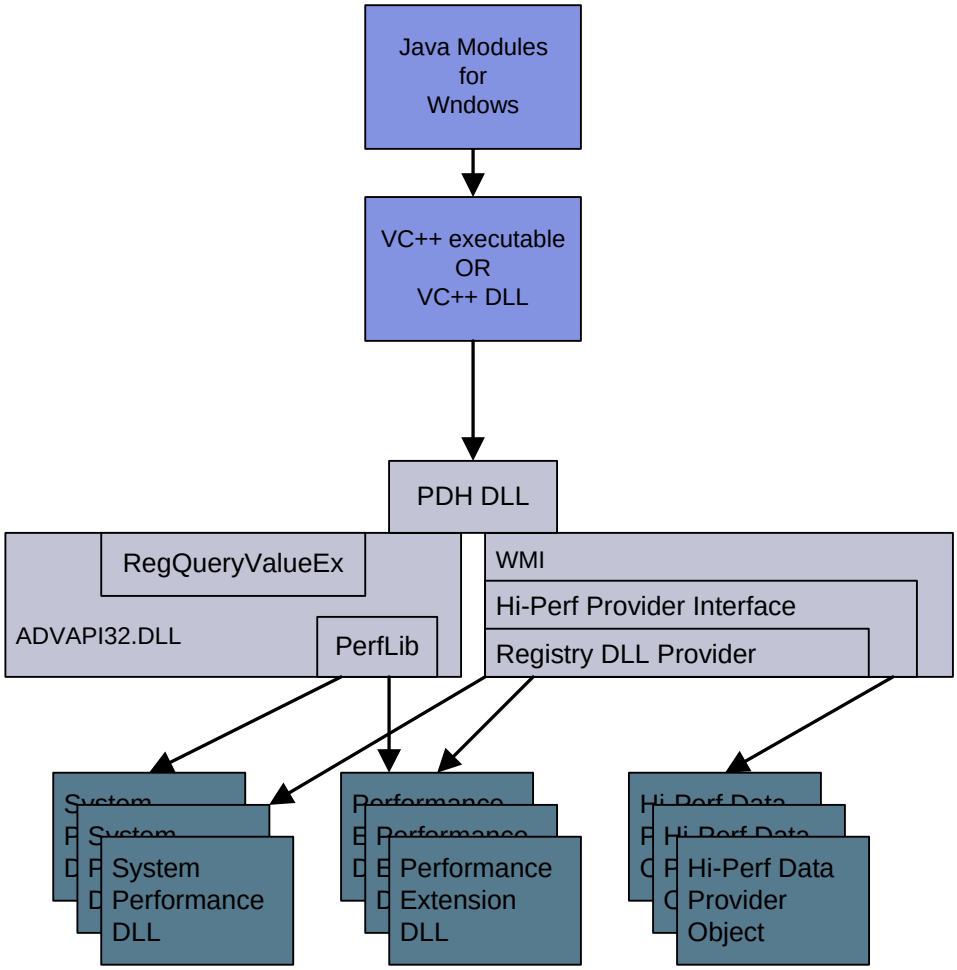

Figure 5.3 – Windows Performance Monitoring Architecture

Data Collection Mechanism

The mechanism used for data collection is the *counter*. A counter is a performance data item whose name is stored in the registry [16]. Each counter is related to some specific area of system functionality. For instance, the processor's busy time, memory usage or number of bytes received over a network connection. Each counter is uniquely identified by its name and its path, or location. In the same way that a file path includes drives, directories, subdirectories, and file names, a counter path consists of four elements: the machine, the object, the object instance, and the counter name. The syntax of a counter path is:

\\Machine\PerfObject(ParentInstance/ObjectInstance#InstanceIndex)\Counter

Here the \\Machine portion is optional; if included, it specifies the name of the machine. If we do not include a machine name, the PDH Library uses the local machine. The \PerfObject component is required; it specifies the object that contains the counter. If the object supports variable instances, then we must also specify an instance string. The format of the (ParentInstance/ObjectInstance#InstanceIndex) portion depends on the type of object specified. If the object has simple instances, then the format is just the instance name in parentheses. For example, an instance for the Process object would be the process name such as (Explorer) or (MyApp). The \Counter portion is required; it specifies the performance counter.

A query is a collection of counters [16]. In order to collect data associated with the counters, queries are created and counters are added to them. Each of these queries can be

individually updated to gather the raw data associated with each counter in the query.

Java Windows Modules

EMA is basically developed in java to achieve the goal of platform independence; therefore some method was required to collect this data from the VC++ application. Modules have been developed which start the executable at the back end and parse the information displayed by the executable using a Buffered reader.

Java Native Interface vs. Executables

Two approaches can be used to collect information from the lower layer. A dynamic link library can be developed and java native interface can be used to call functions and extract performance data from Windows kernel. The second approach is to develop executables giving the required information and then parse it.

Both approaches were adopted during the development of EMA. Experiments showed that JNI had memory leaks. The memory usage constantly increases by a fixed amount and to fix that a garbage collector is required on the cost of CPU usage. Thus we need to have a compromise on memory or processor usage. The executables neither increased the memory nor the CPU usage. Therefore this approached has been followed.

Each module that has been developed for this purpose extends the cmdExec class, which determines that if the operating system is Windows and starts the executable that collects the required data. The Buffered Reader reads one line at a time from the output of the exe (which is continuously displaying performance data values) and stores it in a string. A String tokenizer is then used to parse the values for different parameters. The Windows

modules associate a category and unit with each parameter and return the results to the control modules. There are two java modules developed for gathering the required data i.e monProcSystem and monProcNetwork

The monProcSystem module collects all the performance parameters except those related to the network. The categories that fall under this are CPU, System, Memory and Disk. The category CPU covers CPU Usage, CPU System, CPU User, CPU Idle. System includes Processes, Threads, and Load Average.

Load average values have to be calculated for Windows, as these are not provided directly by the operating system. The concept of load is taken from the Linux Operating System. In Linux, "*The load average numbers give the number of jobs in the run queue (state R) or waiting for disk I/O (state D) averaged over 1, 5, and 15 minutes* [17]." "*It is a damped time-dependent average* [18]." In order to calculate the load on the Windows, the following equation [18] was used

$$load(t) = load(t\text{-}1)\, e^{-5/T} + n\,(1 - e^{-5/T})$$

Where n is the number of processes in ready queue or waiting for disk I/O and T is the time period over which the load average is taken.

Memory category includes Pages in, Pages out, Memory usage, used and free memory. Disk consists of Disk IO, used and free disk space.

The monProcNetwork module is responsible for extracting the Network data from the NetworkIO executable. The first value returned by the executable is the number of Network Interface Cards on the system and based on that value the module formulates the total number of network values to report back to the control module. The parameters retrieved using this module are eth_in which is the rate at which bytes are received at the interface, eth_out which is the rate at which bytes are sent onto the interface, Lo_in which is the rate at which bytes are received at the loopback interface and Lo_out which is the rate at which bytes are sent onto the loopback interface.

5.2.1.1.2 Linux Modules

Linux provides a *proc* directory, which is a subdirectory of root (/). This is where the information about various system resources is stored in the form of files. The user can read these using any command or text editor. Similarly programmers also only need standard file reading procedures for extracting the information.

The interesting thing in this regard is that these are virtual files and are not stored on any hard disk etc. Rather these files are generated from the information stored by the kernel in the memory when users try to access them. So using these files is a highly efficient process.

End-host Monitoring Agent also uses the information stored in this directory. However a shell command is also being used to get the information not available in the proc directory. The only exception for now is disk space usage. Figure 5.4 shows the working of Linux modules.

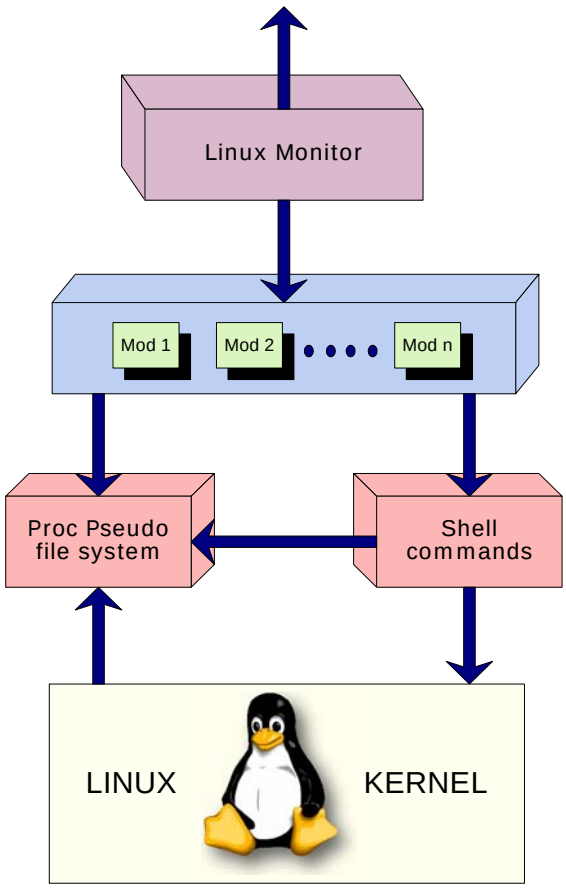

Figure 5.4 - Linux Performance Monitoring Architecture

The information on Linux machine is gathered through six different modules that execute as independent threads.

The monProcVarious module gathers parameters by parsing the contents of /proc/stat file. Extracted parameters include CPU Usage, CPU User, CPU Sys, CPU Idle, Paging Activity, Pages In per second, Pages Out per second, Disk Activity and Blocks (R/W) per second.

The monProcMemUsage module parses /proc/meminfo file for raw data and performs certain simple calculations to get the parameters Memory Usage in percentage, Used Memory in MB and Free Memory in MB.

monProcNetworkIO module parses the /proc/net/dev file and calculates information regarding traffic for various interfaces like Ethernet cards and loop back in Mbps.

The monProcLoad module collects information about the system load from the /proc/loadavg directory. Apart from it, this module also extracts the number of processes information. So this module gathers Load average for 1 minute, Load average for 5 minutes, Load average for 15 minutes and Number of Processes.

monProcDiskStat module collects information about current disk space usage . It executes the standard shell command 'df'. It only gathers info about currently mounted file systems. The parameters extracted by this module are Total Disk Space in GBs, Used Disk Space in GBs and Free Disk Space in GBs.

5.2.1.1.3 Network Monitoring Modules

Monitoring of network paths is one of the fundamental requirements to identify the bottlenecks in overall network performance. Currently large number of tools and techniques are available, which cover this requirement in different domains with varying degrees of success. The modular/layered architecture of EMA facilitated the integration of such tools in this framework with very little effort. This will provide the user with a single platform to carry out testing of various network aspects.

Presently, the network information for EMA is obtained by using Iperf and Abing. Information is collected by parsing values obtained by these tools. This includes bandwidth measurement between host running EMA and Iperf servers, and bandwidth to

and from Abing reflectors. Server/reflector IP addresses are obtained by either setting properties in the JNLP file or by providing the application with these IP addresses at runtime.

Three modules are responsible for network monitoring. The **monPing module** interfaces with the Ping utilty on Linux and Windows. It does this by running the ping command and pings the specified host. The parameters extracted by this module are RTT for remote host, %age of Packet Loss.

Iperf module interfaces with the Iperf tool. Iperf is a tool used for measuring the bandwidth between two hosts. it can do bandwidth calculation for both sides. For a remote host it gives the measures TO Bandwidth and FROM Bandwidth.

Abing module interfaces with the Abing tool. Abing is a tool used for measuring the bandwidth between two hosts. It can also do bandwidth calculation for both sides. For a remote host, it also gives the measures TO Bandwidth and FROM Bandwidth.

5.2.1.2 DATA COLLECTION MANAGER

There are six main responsibilities of the data collection manager:

1. Instantiation of monitoring modules.
2. Data collection from the monitoring modules.
3. Arranging data into a proper categorical hierarchy.
4. Exposing a simple yet elaborate interface to the GUI, so that the GUI can easily fetch dynamic data for display.

5. Controlling the reporting mechanism.
6. Cleaning on exit.

Instantiation of Monitoring Modules

As two sets of monitoring modules exist, a need for dynamic detection of operating system is required, so that only the appropriate set of modules is loaded at a time. Keeping in view this requirement, the instantiation of modules is implemented through polymorphism. The steps that explain the mechanism are

1. The control engine detects the underlying operating system.
2. Once the operating system is detected, and it is one of the operating systems supported by EMA, the appropriate monitor is instantiated.
3. The monitor is responsible for controlling modules. There are two monitors; one for Linux and the other for Windows. Both implement the same interface and thus are instantiated using polymorphism.
4. Once a monitor is instantiated, it instantiates the monitoring modules in turn.
5. Monitoring modules then start gathering data.

The above steps represent a three level hierarchy *i.e.,* the control engine to the monitor, and then from the monitor to the monitoring modules.

Data collection from the monitoring modules

Each monitor exposes a method that returns an array of monitoring modules. The control mechanism simply fetches all the monitoring modules. Results from each module are extracted and sent to the information repository for storage. This process is carried out in a separate thread.

Arranging data into a proper categorical hierarchy

The information repository is a hierarchy of storage structures that are arranged into categories. Once result is collected form a module, it is sent to the repository in four parts *i.e.,* categories, parameter names, parameter values and their units. The repository on receiving this information validates the data so that no garbage or illegal values enter the repository and keeps each parameter into the corresponding category.

All this is done in a separate thread that is started by the control engine. This thread lives as long as the application keeps running. As the repository is responsible for dealing with data being refreshed after regular intervals of about 3 seconds, this part needs to be efficient in utilizing the memory. To achieve this objective, only one copy of a parameter is maintained in the memory at a time, and latest values are just updated rather than appended. Therefore, at any instance in time, information in the repository basically depicts the latest system state at that time.

The hierarchical structure of the repository can be understood by looking at its architecture. Figure 5.5 shows the architecture:

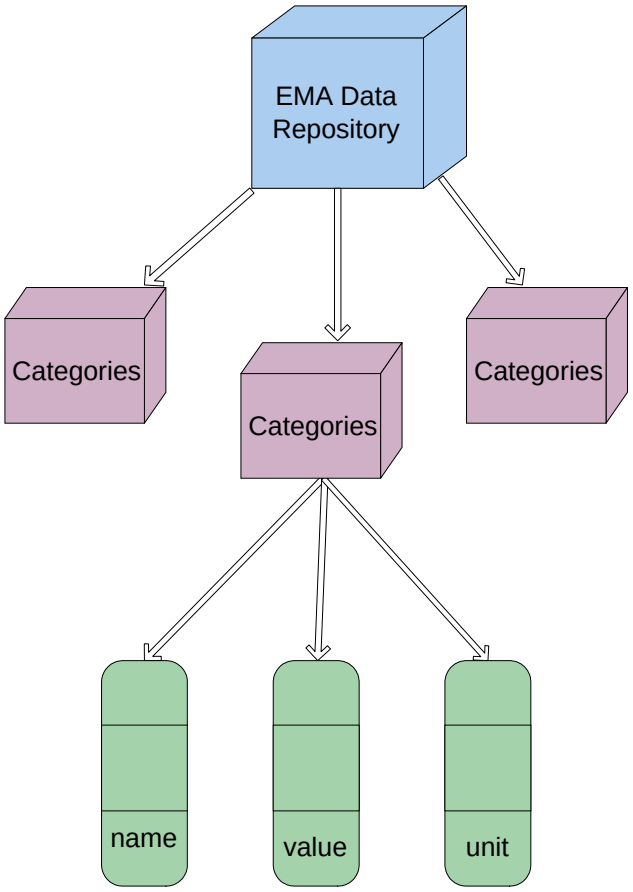

Figure 5.5 – Data Collection Manager

At the top most hierarchy, a list of *Category* objects is maintained. Each category object consists of three lists each; one for parameter names, the other for parameter values, and the last one for units. So each category object represents a complete category, with all the available parameters and their information. Once a parameter is sent to the repository by the control engine, it is first checked if it already exists. If the parameter already exits, its previous copy is deleted, and the latest information is added after validation. If on the other hand, this parameter is found to be new, it is simply validated and added to the appropriate category object. This allows not only for efficient memory utilization, but also for easy addition of new parameters and categories to the system.

The information repository had been tested thoroughly for memory leaks. All the tests carried out have shown that it is extremely reliable, and no memory leaks have been reported yet. Furthermore, as discussed in later, the repository may be accessed by two threads at the same time. For this reason, the data retrieval interface has been provided with synchronized methods. Only those methods have been synchronized where there is a possibility of simultaneous access by two threads. This ensures that no overhead is created for other normal methods; as well as keeping it safe for multiple threads.

Feeding Data to the GUI

To enhance the performance, a single thread has been made responsible for both collecting data from the monitoring modules, and feeding it to the GUI. The sequence for this operation is such that first data is submitted to the repository, and then the GUI fetches the organized data from there. Interface to the GUI is simple enough to cater for all the requirements by GUI. This enables dynamic addition of parameters and classes to the GUI. Categories and parameters are not fixed; the GUI adjusts itself accordingly, mainly because of the underlying flexible architecture.

Controlling the reporting mechanism

The reporting mechanism is also instantiated by the control mechanism. It is started in a separate thread. Once this thread is started, it can report values to the remote server for as long as the application keeps running.

Apart from instantiation, the control engine also interacts with the reporting mechanism through the information repository. To report information to the remote server, the reporting mechanism uses the information repository. The same interface as of the GUI is

used to fetch the information.

5.2.1.3 GRAPHICAL USER INTERFACE

EMA has a graphical user interface, which aims at displaying monitored information to the user in a simple, yet elaborate manner. This is done by providing the user with values along with real time graph plotting which includes both bar graphs and pie charts. For network parameters, bar charts are plotted for each update along with the previous measurements.

Design of the GUI is made scalable in such a way that each attribute is added in a separate panel and the addition of new modules can be adjusted without any extra effort.

5.2.1.4 STATISTICS REPORTING

EMA has a built-in reporting mechanism, to report the gathered performance statistics to any performance data repository. The design includes a server which operates at the receiver end and a client for collecting, organizing and sending the data to multiple servers. The design is described in figure 5.6. The EMA data collection manager starts a separate thread for reporting the information after the given time intervals. All the necessary statistics are packed in a UDP packet and sent to the servers whose IP addresses are specified. Each server receives the packets from multiple clients and uses a thread-pool to process them for optimal performance. The processed values are then supplied to the data repository in the specified format. Since overall design of EMA is modular and very scalable, it can be integrated with various repositories requiring different types of formats without affecting the other modules.

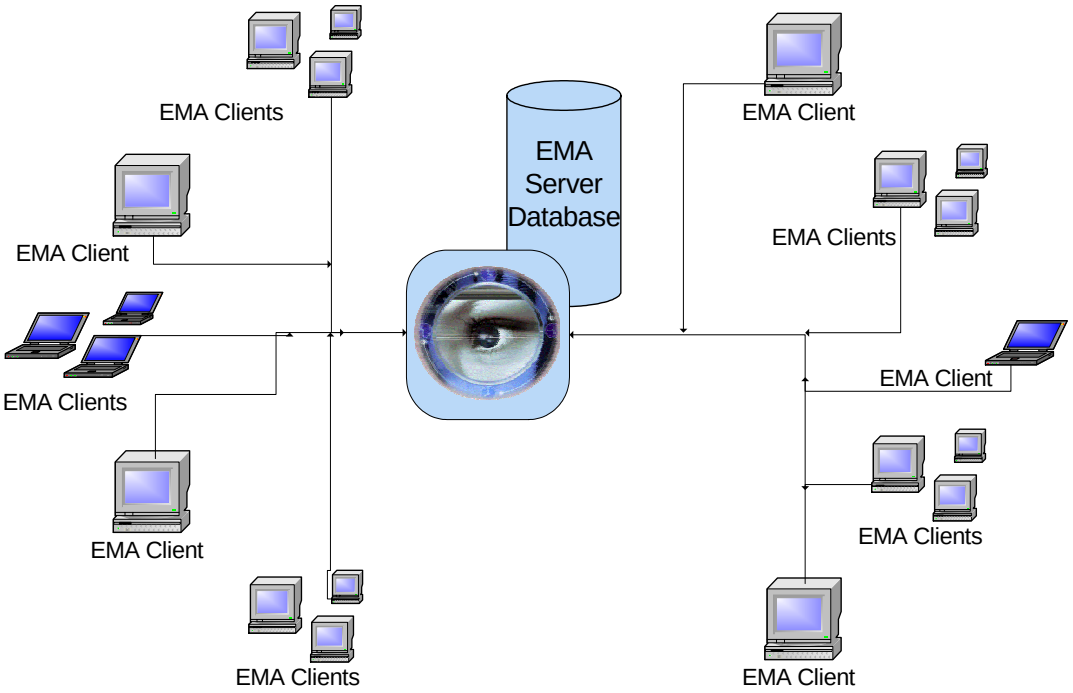

Figure 5.6 – Statistics Reporting

The EMA Client reports values to the EMA Server after every 90 seconds by taking the average of last thirty values. Since the updating of each value takes 3 seconds, the total time spent in calculating the average of 30 values is 90 seconds. According to the Central Theorem [19], if number of samples in a sample exceeds 30, the distribution that generates the random variable (parameters to be determined) approaches Gaussian. Since $\mu_s \pm 1.96\ \sigma_s$ correspond to 95% confidence intervals, we have 95% confidence that the true values of the parameter to be estimated lies in the above interval. Here μ_s is the mean σ_s is the standard deviation

EMA client can report values to its own repository as well as any other distributed repository, with which the EMA Server is integrated, like the MonALISA. Presently

EMA server is integrated successfully with MonALISA[20] and is available at monalisa.niit.edu.pk.

5.2.2 EMA Server Architecture

EMA Server is a UDP Server that receives data packets on a specified port. UDP is a faster protocol as compared to the TCP. UDP, which isn't connection-oriented, is more appropriate for sending limited amounts of data per packet. In case of EMA it does not matter much if a packet is lost, because packets are continuously transmitted after an interval of 90 seconds. The Server Architecture is shown in the figure 5.7.

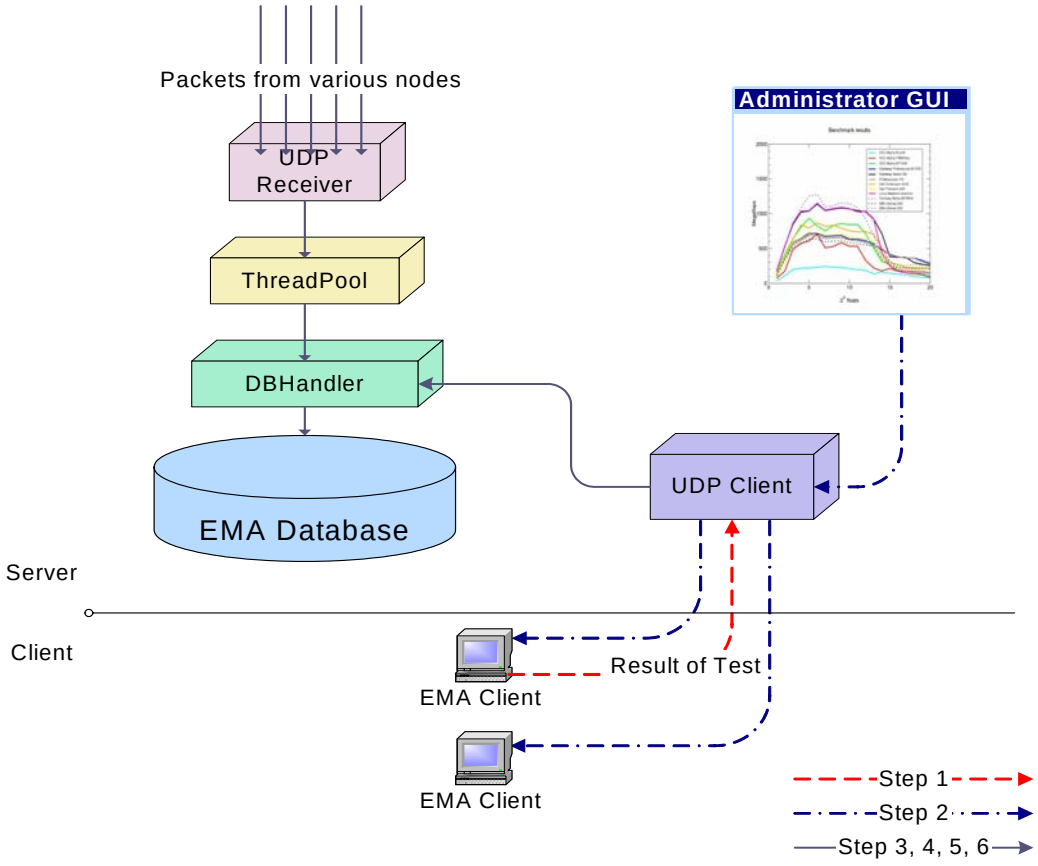

Figure 5.7 – EMA Server Architecture

5.2.2.1 THREAD POOL

The packets received by the EMA Server are processed in a thread pool. As soon as a packet arrives it is passed over to a thread where all the information is extracted, organized and added to the database. The thread pool is initialized to process ten threads at a time, rest of the packets are added to the waiting queue.

Advantages of using a thread pool

Many server applications, such as Web servers, database servers, file servers, or mail servers, are oriented around processing a large number of short tasks that arrive from some remote source. A request arrives at the server in some manner, which might be through a network protocol (such as HTTP, FTP, or POP), through a JMS queue, or perhaps by polling a database. Regardless of how the request arrives, it is often the case in server applications that the processing of each individual task is short-lived and the number of requests is large.

One simplistic model for building a server application would be to create a new thread each time a request arrives and service the request in the new thread. This approach actually works fine for prototyping, but has significant disadvantages that become apparent while deploying the server application. One of the disadvantages of the thread-per-request approach is that the overhead of creating a new thread for each request is significant; a server that created a new thread for each request would spend more time and consume more system resources creating and destroying threads than it would do processing actual user requests.

In addition to the overhead of creating and destroying threads, active threads consume

system resources. Creating too many threads in one JVM can cause the system to run out of memory or thrash due to excessive memory consumption. To prevent resource thrashing, server applications need some means of limiting the number of requests being processed at any given time.

A thread pool offers a solution to both the problem of thread life-cycle overhead and the problem of resource thrashing. By reusing threads for multiple tasks, the thread-creation overhead is spread over many tasks. As a bonus, because the thread already exists when a request arrives, the delay introduced by thread creation is eliminated. Thus, the request can be serviced immediately, rendering the application more responsive. Furthermore, by properly tuning the number of threads in the thread pool, you can prevent resource thrashing by forcing any requests in excess of a certain threshold to wait until a thread is available to process it.

5.2.2.2 DBHANDLER

The DBHandler serves as an interface between the server and the database. It provides all the methods to connect to the database, create queries and execute them. The information that is to be added to the database is first passed to the DBHandler, which checks if the target tables exist. If new tables have to be created, it generates the requisite queries and requests the database for them.

The DBHandler is also the interface between the administrator GUI and the database. In this case it is used to retrieve the required information from the database.

5.2.2.3 DATABASE

The database keeps the record of the static and dynamic data reported from each host as well as the network monitoring results obtained as a result of remote monitoring. There is a table for static data for all the clients connected to the EMA server and it is updated each time the client connects or re-connects to the server. A dynamic data table is created for each client connected to the server and values are added after specified time intervals whenever the client reports latest performance data.

5.2.2.4 REMOTE MONITORING

An administrator GUI is provided at the server side to monitor all the connected nodes as well as to conduct network performance measurement tests between any two nodes. These tests are conducted on demand and the parameters that are measured for the network path are Bandwidth to, Bandwidth from, RTT and Packet loss.

These tests are conducted using the tools Iperf and Ping.

5.3 Object Model

The design of EMA is split into the three packages.

i) Monitor Package

ii) GUI Package

iii) Util Package

5.3.1 Monitor Package

This is the most significant package of EMA architecture. It is where most of the business logic of the Application resides. It contains standalone classes as well as sub packages.

The Object Model is given in figure 5.8.

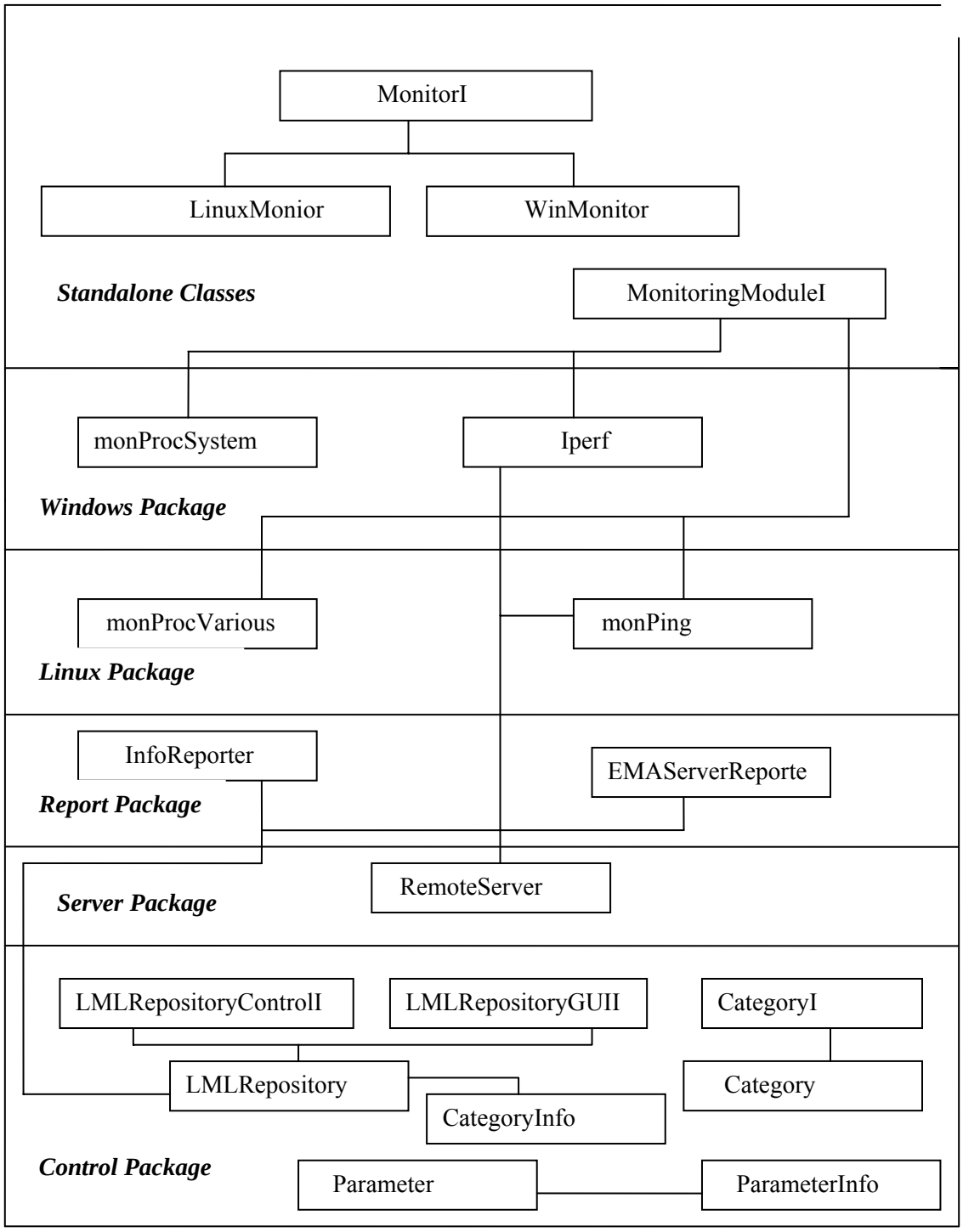

Figure 5.8 - Monitor Package

MonitorI is an interface. It is a base for developing monitoring control classes for various platforms like Windows and Linux.

WinMonitor is a 'controller' class which controls all the underlying monitoring modules for windows platform. It implements the MonitorI interface. It starts the monitoring modules, gathers static parameters and performs other host specific functions. While the LinuxMonitor is also a 'controller' class and has same functionality as WinMonitor. It also implements the MonitorI interface. It is to be noted that either WinMonitor or LinuxMonitor is loaded in JVM depending upon the underlying OS.

MonitoringModuleI is an interface. It defines the behaviour of a typical monitoring module. All the monitoring modules, either for Linux or for Windows, must implement this.

The Win Package is a subpackage of Monitor package and contains classes which act as monitoring modules for Windows. It is these modules which extract the performance parameter information by interacting with the OS.

The Linux Package is also a sub package of Monitor Package. It contains classes, which act as the monitoring modules for the Linux platform. The monitoring mechanism for Linux and Windows is significantly different. But these classes present the same interface to the upper layers of the EMA architecture, thus hiding the underlying heterogeneity. It makes the upper layers of the EMA platform independent.

Control Package is a sub package of the monitor package, and forms the control layer of EMA. It contains the controller classes that gather information from the underlying monitoring modules and maintain a runtime repository of the performance data. It is this repository with which the GUI interacts for presenting information to the user. The reporting mechanism also interacts with this repository for reporting the values to the central server.

The Report Package is a sub package of the Monitor package. It handles the reporting of gathered data to the central server using the UDP sockets.

The Server Package is the package that handles communication with the central server. It spawns a TCP server. Via a protocol, similar to HTTP, EMA server sends control commands to it. These commands can be used for Remote Testing.

The Util Package contains the general purpose classes, which are needed through out the code.

The GUI package for the handling of GUI on the client side of EMA. It has various classes to handle different aspects of the GUI. It has methods to obtain data from the LMLRepository and to update data for the user in the GUI. The GUI has two ways of updating the data. The Data is passed on by the control classes to update the data or the Data is requested on demand by the user and as a reaction, the GUI is updated.

5.4 Interface Design

The GUI (Graphical User Interface) of this project has been carefully designed keeping in view the needs of the end-user as well as the central managing server. There are two main interfaces which make up the EMA Client side GUI and Server side GUI.

5.4.1. Client Side GUI

The Client side GUI is composed of two main JPanels. These are:

Static Data Panel which Panel displays the static data of the client. This is a static panel and is not dynamically updated. It is shown in figure 5.9.

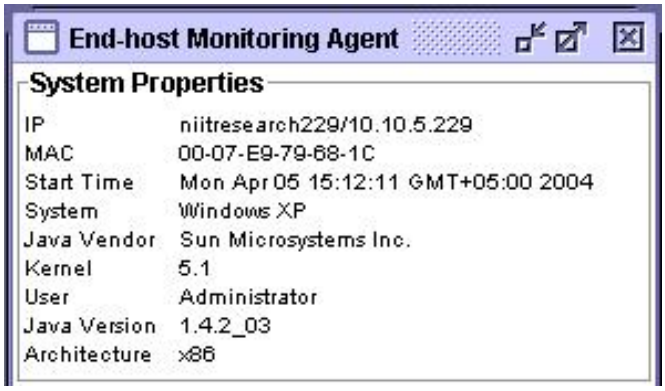

Figure 5.9 – Static Data Panel

Dynamic Data Panel consists of three main Panels. They are The Host Statistics Panel, shown in figure 5.10, is continuously updated with the generation of values. It contains CategoryPanel objects, which in turn have NameValuesPanel objects. This panel has graphs and pie charts for different parameters and categories, which are updated every three seconds.

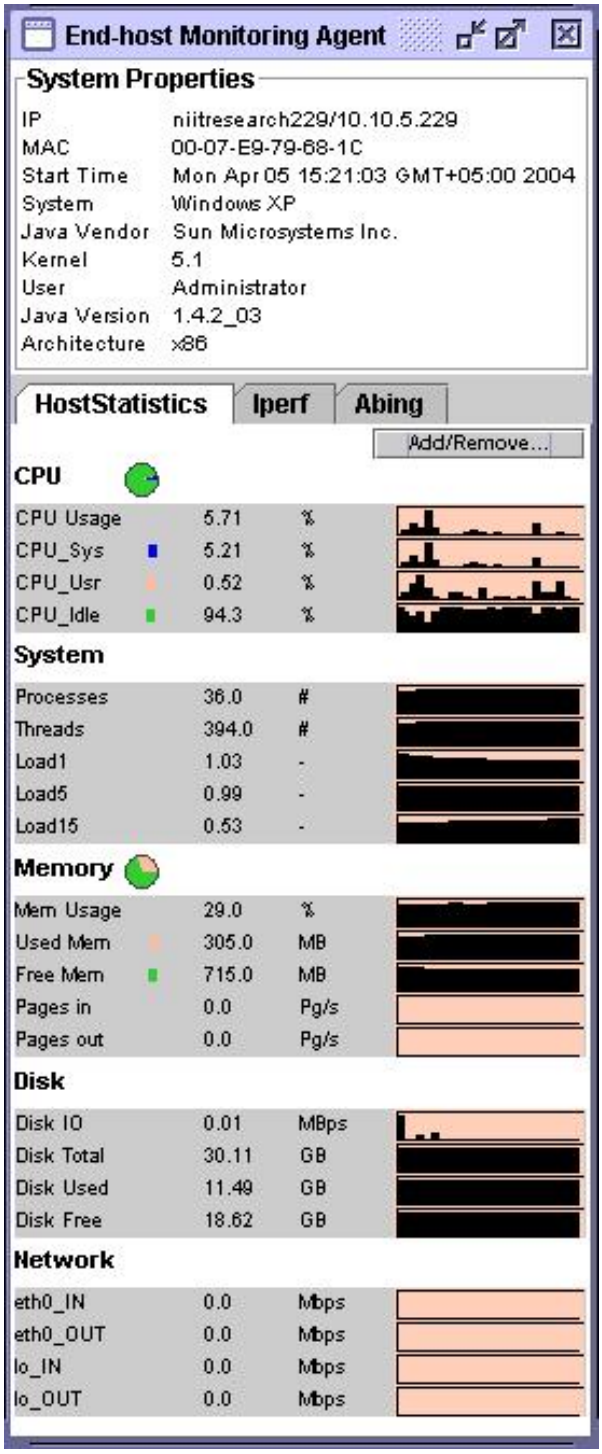

Figure 5.10 – Host Statistics Panel

There is a customizable panel, shown in figure 5.11, using this visible parameters can be increased or decreased, with the aid of the Add/Remove button. By default, all the parameters are selected.

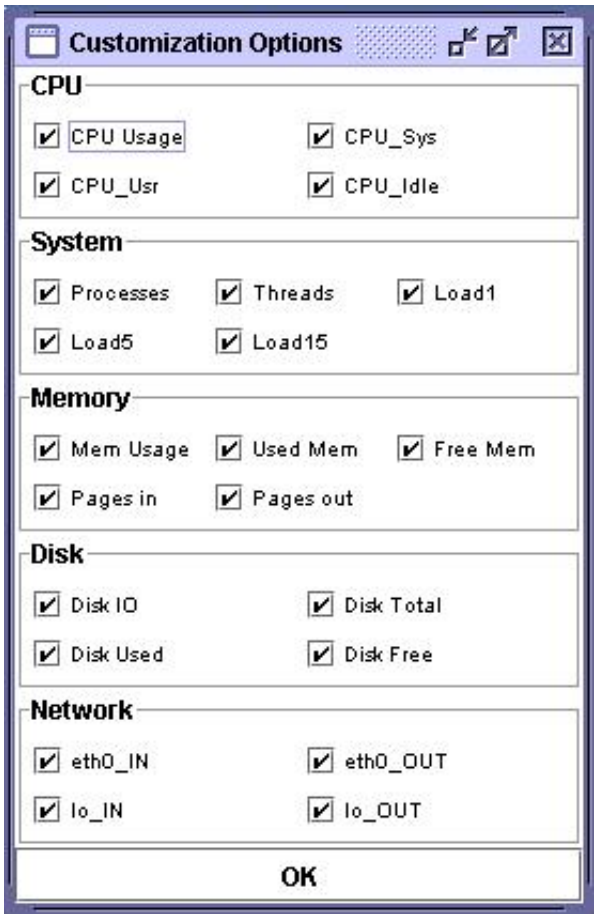

Figure 5.11 – Customization Options

Iperf Panel contains upper and lower Panel and these again contain sub-panels for graphical display of on-demand bandwidth tests conducted by the user. It shows graphs for one-way bandwidth and compares with the previously obtained value in graphical notation, which makes it easy for the user to deduce network performance level. Figure 5.12 displays the Iperf Panel.

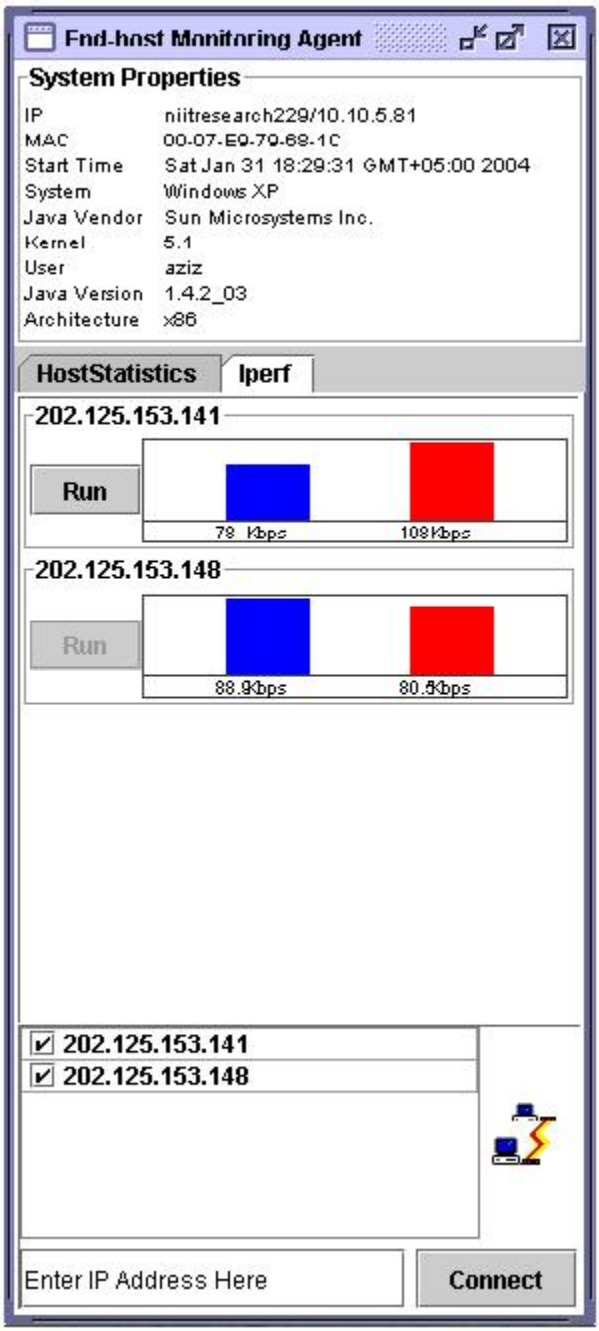

Figure 5.12 – Iperf GUI

The ABing Panel consists of sub-panels, which display results obtained after the on-demand tests, initiated by the user. It displays plots for two-way bandwidth, *i.e.*, "Bandwidth To" and "Bandwidth From" a particular node, specified by the label URL.

Again, it makes comparisons with previous measurements, thus giving an overview of the overall network performance.

5.4.2 Report GUI

The report GUI shown in figure 5.13 is used to select the central servers to which the EMA client is reporting. There can be one, two or many central servers to which the EMA client reports. They can be added or subtracted and dynamically allocated.

Figure 5.13 – Report GUI

5.4.3 Server End GUI

The server end GUI shown in figure 5.14 comprises of an initial GUI, which consists of a list of IPs that have entries in the database of the EMA Server. The historical values are displayed at one end and their graphical plots are made. This gives the remote machine an idea of the performance of the client machines.

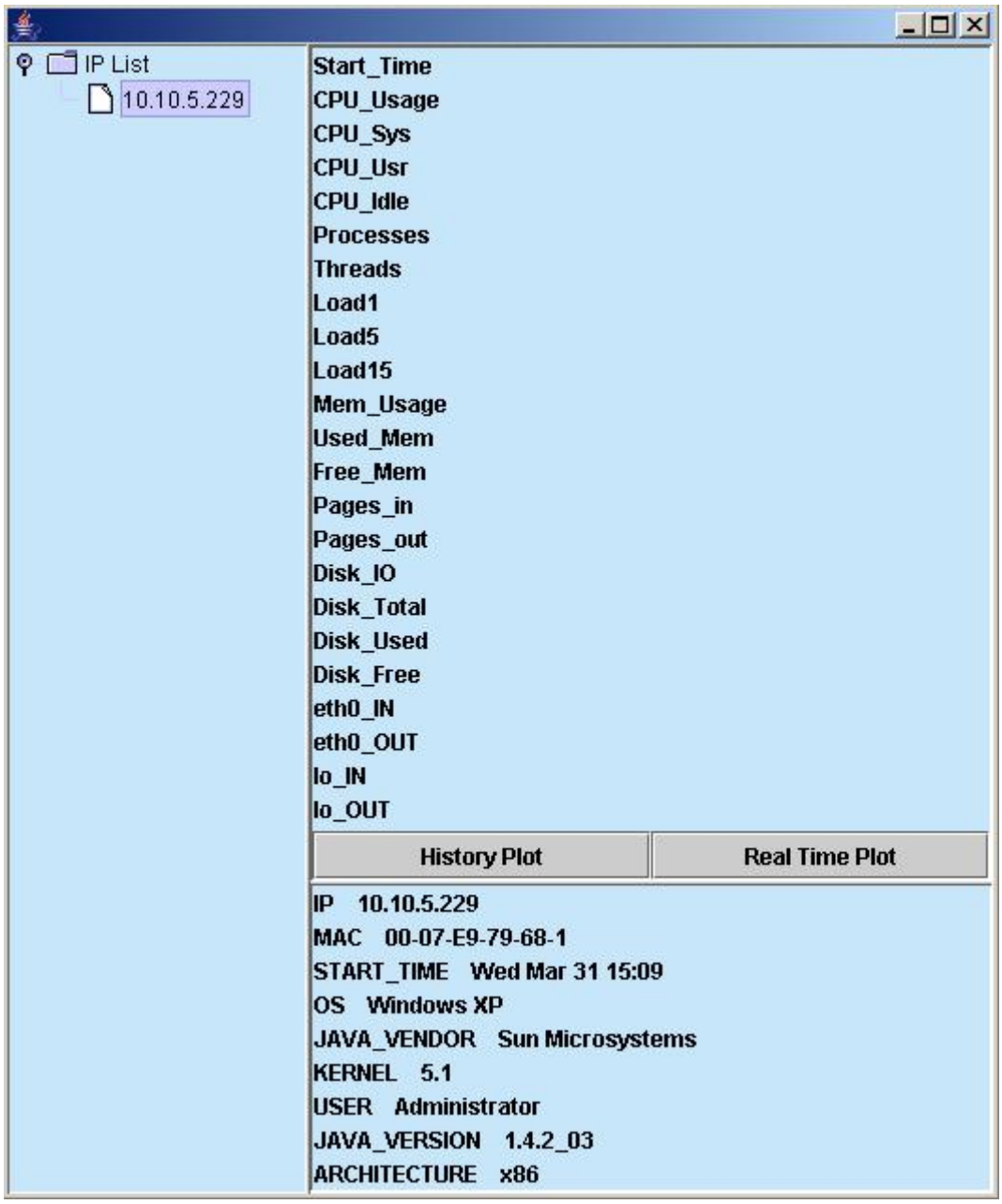

Figure 5.14 – Administrator GUI

The plots are drawn using the Java Analysis Studio (JAS). They are shown in two main forms, one in the form of histograms and other in form of line graphs.

5.4.4 Histograms or Bar graphs

Histograms and bar graphs can be viewed to analyze the history and trends of performance parameters reported from different EMA Clients connected to the Server. The values are plotted against time at which the measurement was taken. These are shown in figure 5.15.

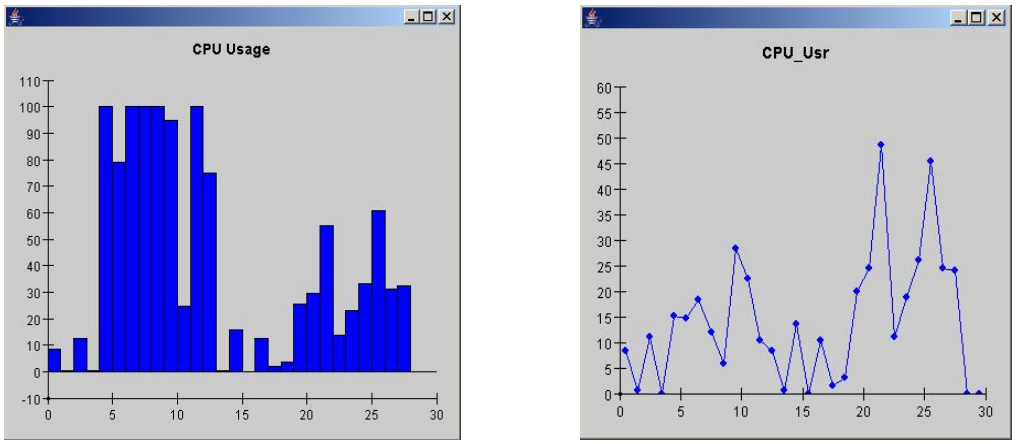

Figure 5.15 – Histograms and Line Graphs

5.4.5 Remote Network Monitoring GUI

The remote network monitoring GUI shown in figure 5.16 gives the user an option to select any two remote IPs and perform a network test. The result is displayed after comprehensive testing in form of a report. The instantaneous report can be viewed, which gives the user an idea of the network state.

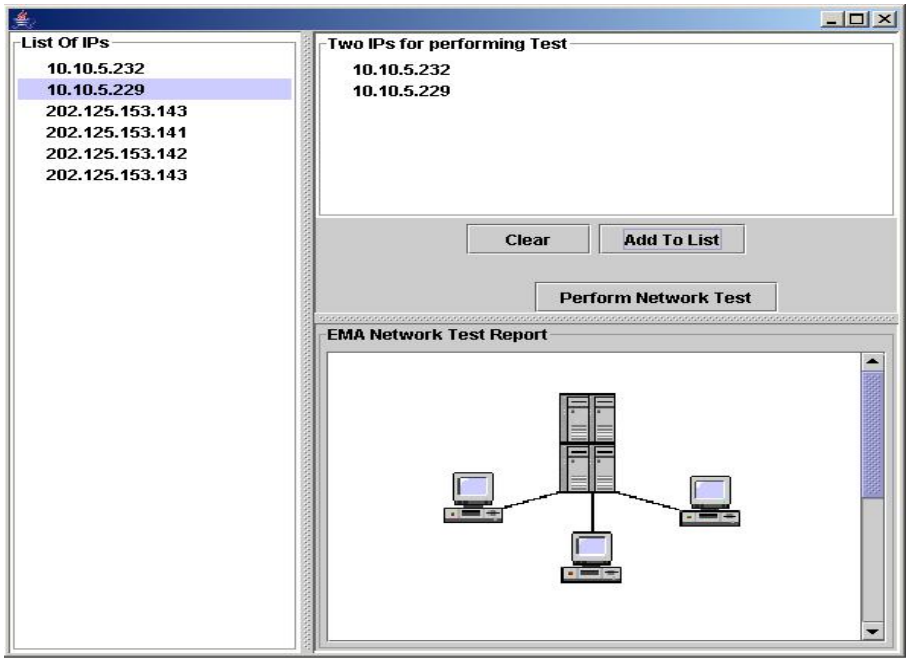

Figure 5.16– Remote Network Monitoring GUI

5.4.6 Historical Reports

The user can also view previous reports from EMA's network history bank. These reports are kept as HTML files in the history folder. These reports can be extremely valuable for the user as network administrators can compare the network statistics based on these reports. The name of the generated HTML file denotes the two IPs amng which the test is conducted and this is concatenated with the time of report, which makes it easy for the administrator to sort out a particular report.

6

TRANSITION

The transition phase includes all the activities, after construction, that led to the final product and documentation to NUST and CERN. Following activities took place during this phase:

6.1 Deployment

The client application is available in two versions:

1. A dynamically downloadable application that can be downloaded from http://202.83.166.180/EMA/ema.jnlp and http://monalisa.cern.ch/ema.jnlp

2. Application that can run locally using scripts.

A server integrated with MySQL Database and administrator GUI. Scripts for installing all these are provided.

6.2 Beta Testing

Testing and validation are one of the most important phases of any development activity. Testing was carried out throughout during the development process. An iterative methodology was adopted as a whole: each phase of development was visited numerous times, testing implicitly being part of each phase.

6.2.1 Testing Environment

EMA was tested on various machines, and a comprehensive performance analysis was carried out. The testing environment consisted of 100 Mbps LAN with the following

types of machines connected to the network. We have used different platforms to ensure platform independence.

	Processor	RAM	OS
PC I	2.4 GHz	1.0 GB	MS Windows XP
PC II	2.4 GHz	1.0 GB	MS Windows 2K
PC III	2.0 GHz Xeon	900 MB	Linux Redhat 7.3

Table 6.1 Testing Environment

6.2.2 Software Testing

This phase dealt with software testing, which essentially required the system to be executed and hence tested in different ways for different scenarios.

6.2.2.1 Defect Testing

The aim of defect testing is to make the software behave incorrectly in different situations. This helps in identifying various flaws as well as constraints related to user interaction with the system.

In Functional/Black Box Testing the system is considered to be a black box and its behavior is determined by studying its inputs and related outputs. Black box testing was not only carried out just for the whole system but also for the individual components. For this purpose different types of test data were identified and tested.

Structural or white box testing is used for testing small pieces of code. It was applied successfully and we were benefited especially for the algorithm for load calculation on Windows.

6.2.3 Stress Testing

Once all components were completely integrated, stress testing was carried out to determine the robustness and reliability of the software, the CPU usage consumed by both the client and the server, the load they exerted on the system, the amount of memory consumed etc. The results showed EMA to be a light weight application and it had to be since it was meant for performance monitoring. This established that the design is not only reliable but also efficient.

6.2.4 Software Inspections

This phase dealt with the testing of the software, which does not require execution of the system.

Testing of the design of the system was implicitly exercised by the following iterative approach of RUP. Individual modules were first identified, designed, assessed and then put together to get a bigger picture.

Regular dry runs of modules were carried out to ensure that the implementations were fool-proof.

Control Flow Analysis was carried out for the verification and validation of control blocks in the source code were carried out, for instance, the ‘for’ and ‘while’ loops and the ‘if’ condition blocks.

Data Use Analysis was done to find and remove improper initializations, unnecessary assignments and the variables that were declared but never used.

Interface Analysis was used to ensure consistency of interface, class and procedure declarations, definitions and their use. It was observed through tests that all the methods declared in the interfaces were correctly implemented in the classes and that there were no redundant methods.

7

RESULTS

7.1 Performance Results

The aim of performing these tests was to measure the performance of EMA. In this case, the total CPU time consumed by the EMA Clients was recorded for an hour. The CPU time consumed by EMA gives a measure of its lightweight nature. The following pie-chart gives a comparison of average of the total CPU times consumed by EMA modules on different machines.

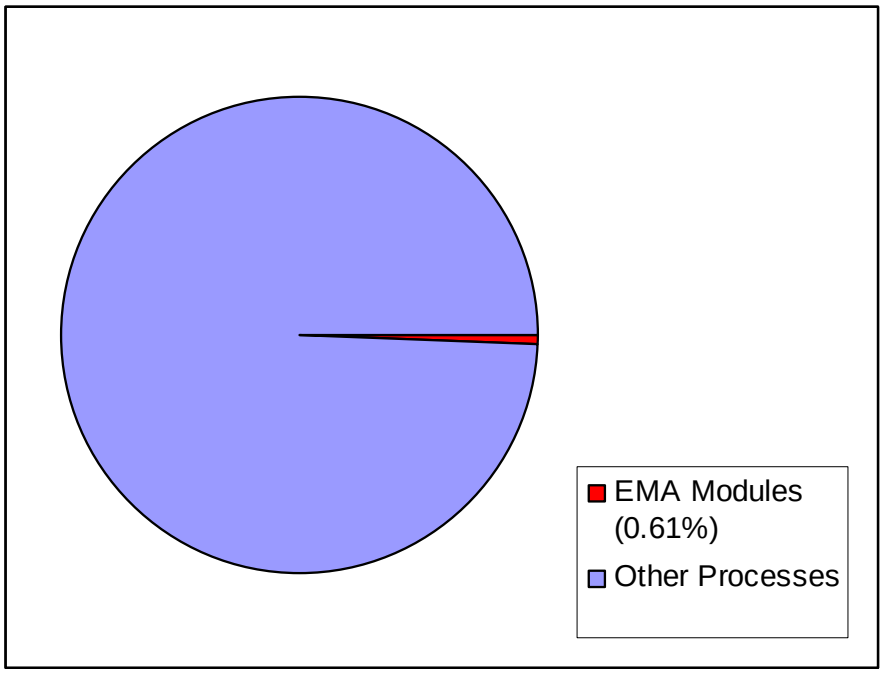

Figure 7.1 – CPU Time Consumption

An overview of the instantaneous CPU usage by EMA is given in the following plot. The spikes, which occur after every 3 seconds, shows that the CPU is only consumed when the values are retrieved from the OS kernel and simultaneously updated in the GUI, which updates the plots and pie-charts for different parameters. This behavior was only observed on the Windows operating System. On the Linux there were no spikes.

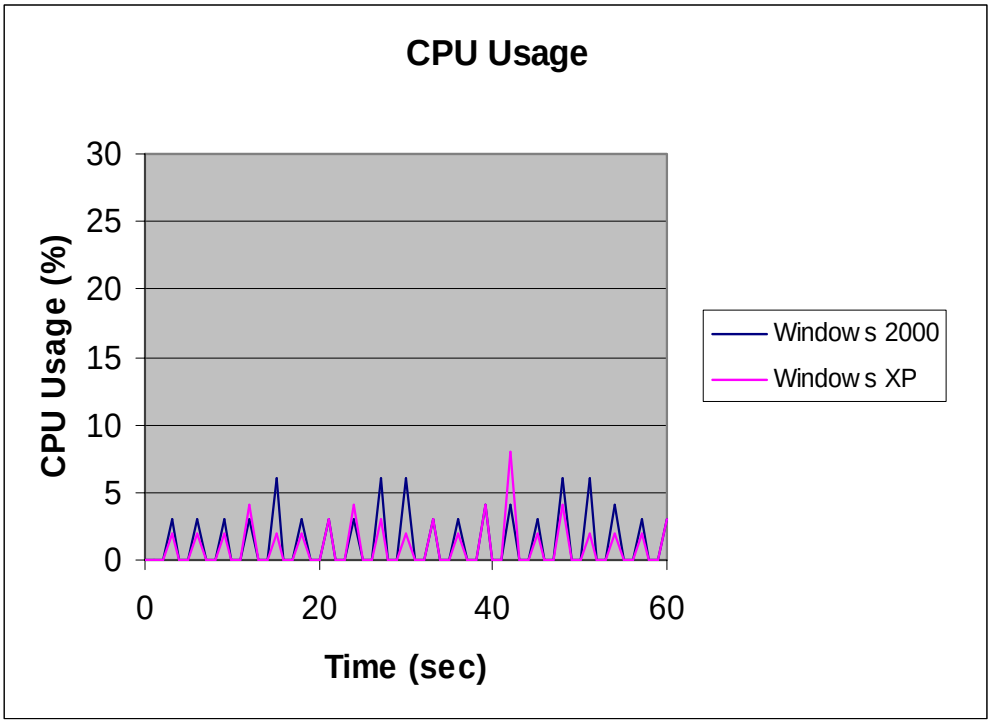

Figure 7.2 – CPU Usage Behavior

7.2 Network Traffic Analysis

Since each EMA client sends 684 bytes of data in a UDP packet to the server after 90 seconds, it does not congest the network. Even in the worst case scenario of 100 clients, when they send packets at the same time, the total number of bytes generated is 68400 after 90 second intervals. This makes the EMA an extremely light weight application as far as network traffic is concerned.

7.3 Load Calculation Results

There is no provision of Load on Windows so it was calculated using the load calculation method in Linux described by Dr. Neil Gunther. (For details see Appendix D). According to his results for a hot loop the 1-minute samples track the most quickly while the 15-

minute samples lag the furthest. The behavior of load average is shown in figure 7.3

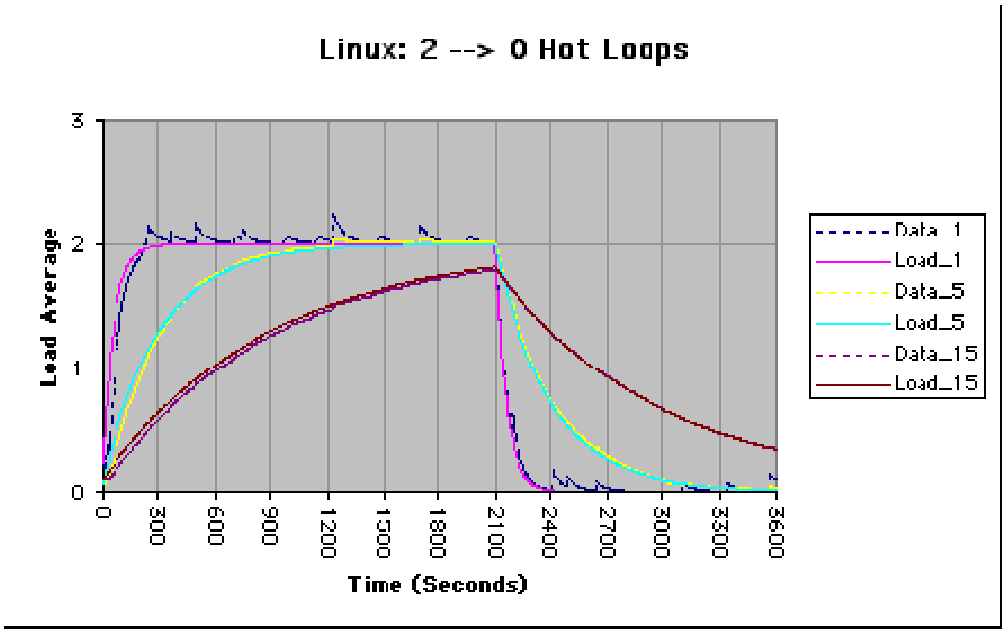

Figure 7.3 – Load Average Trends

In order to validate our findings and the load calculation method we also conducted tests on Windows and Linux. The trends observed were very similar to those in figure 7.3. The results for Linux are shown in figures 7.4 and 7.5 and for Windows in figures 7.6 and 7.7.

The behavior of Load on Linux for a hot loop was noted against the CPU Usage and same was done for windows. The behavior on Windows was similar.

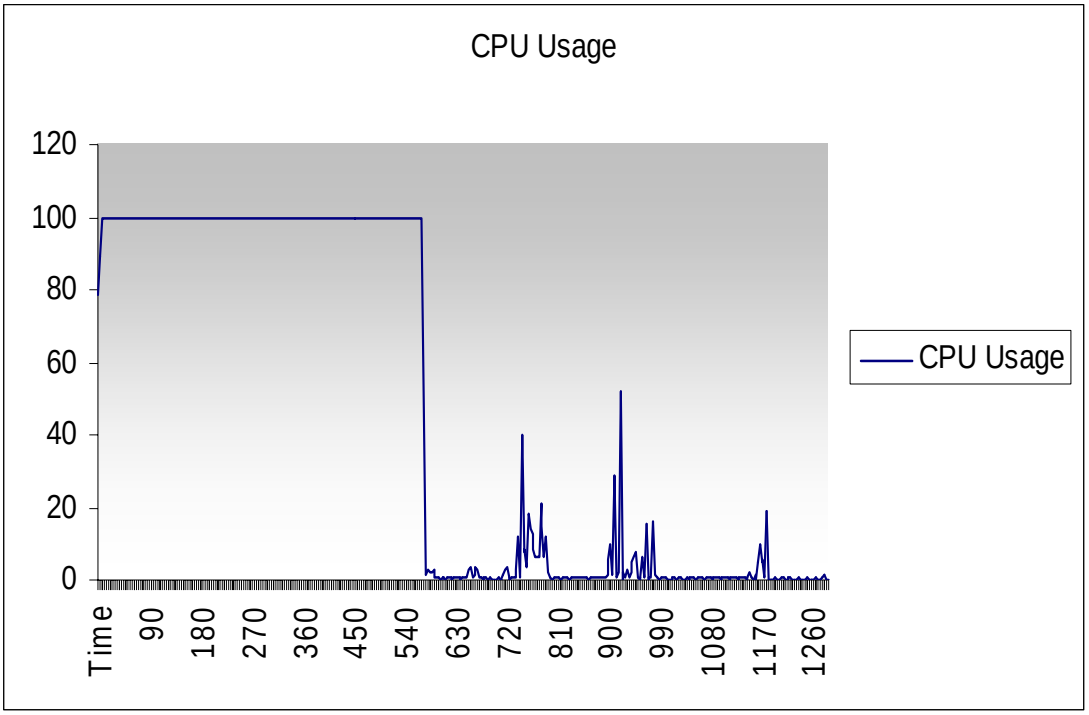

Figure 7.4 – CPU Usage on Linux

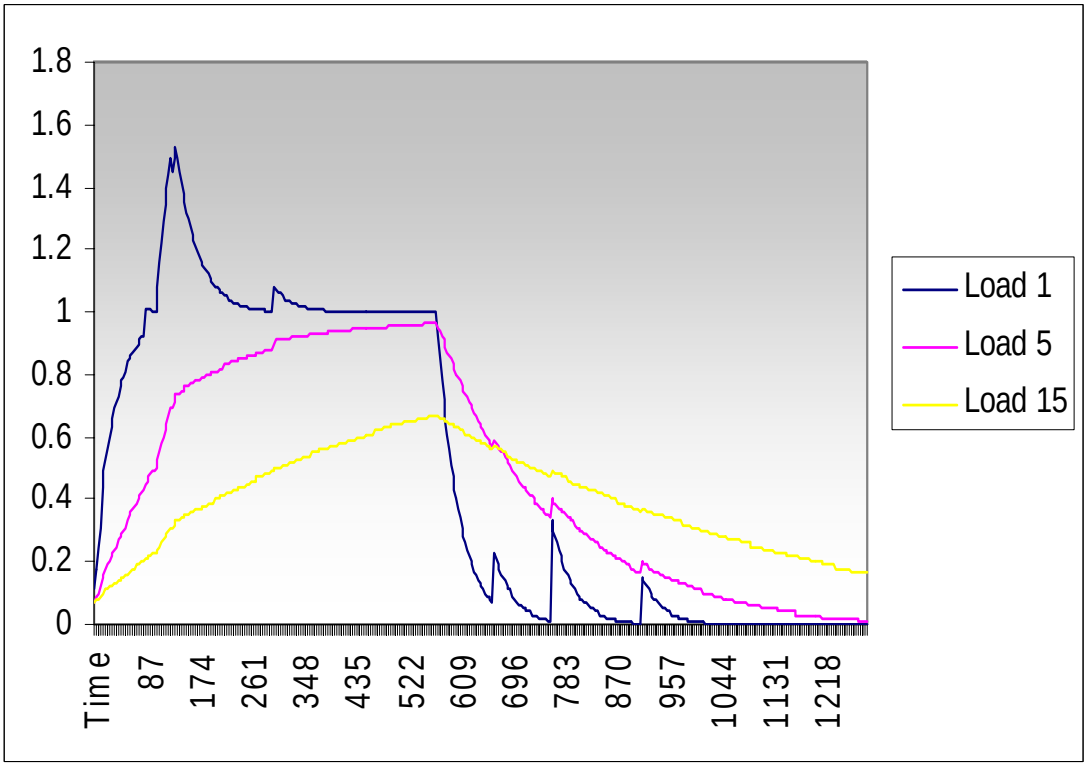

Figure 7.5 – Load on Linux

In case of Linux the Load Average for 1, 5 and 15 minutes does not normally rises greater than 2 and the value of load greater than 2 indicates congestion but on Windows the load Average went to 10 although the rise and fall trend was the same. The reason is the difference in Operating System architecture. In case of Windows the maximum size of processor queue length is 10 but for Linux it is 2. As in Windows a sustained processor queue length of greater than four threads generally indicates that a single processor is doing some real work, although such a value does not necessarily indicate a performance bottleneck. Hence value of load average on Windows was larger than that on Linux.

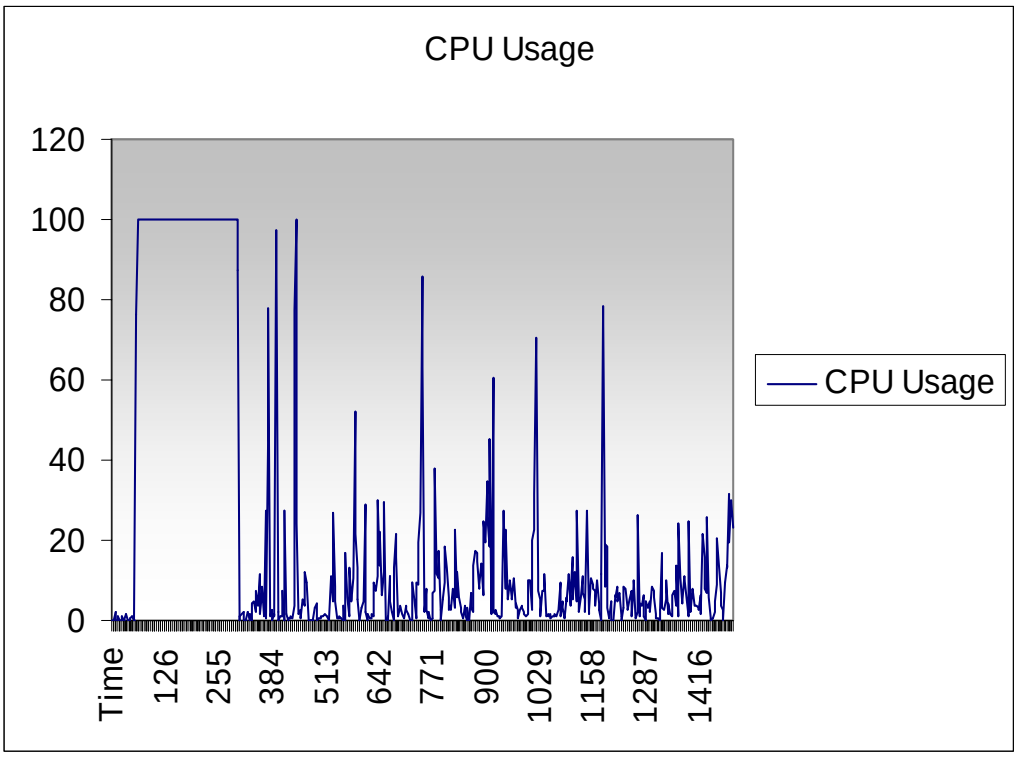

Figure 7.6 – CPU Usage on Windows

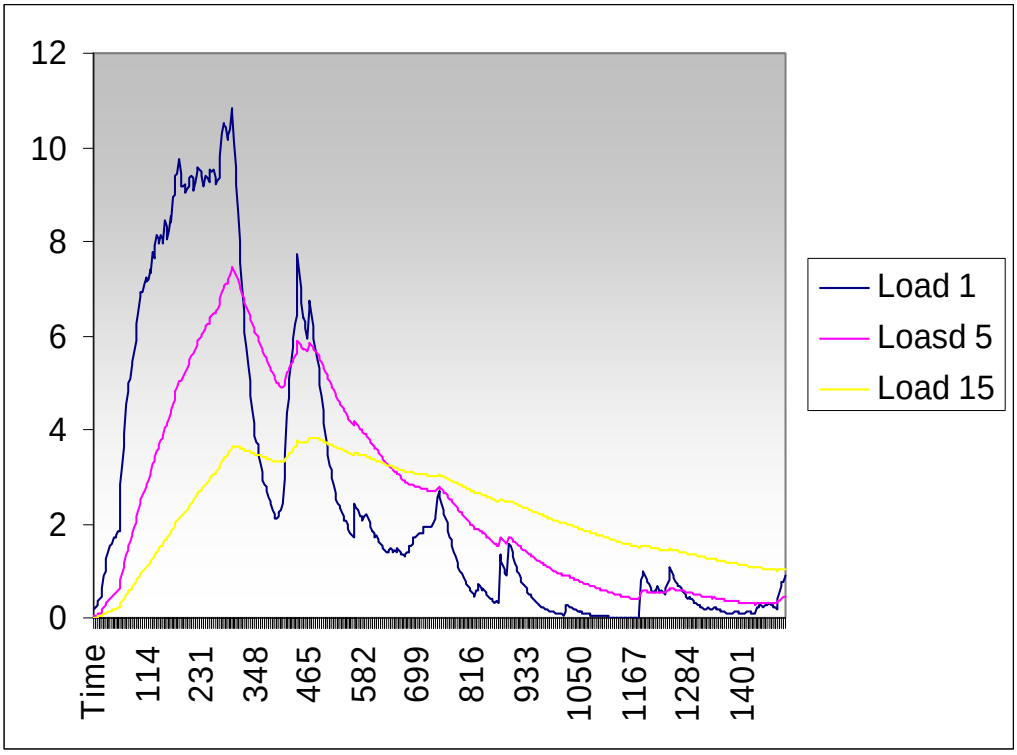

Figure 7.7– Load on Windows

8

CONCLUSION

8.1 Conclusion

EMA brings along the concept of a decentralized resource monitoring architecture. It applies not only to situations where network diagnostics is required, but also to any other scenario, where there is a need for resource monitoring. This is due to the fact that EMA bears the capability of reporting the collected information to remote servers, where analysis may be performed in order to make job scheduling decisions.

One of the major alternatives to this architecture is the use of SNMP as the information retrieval mechanism. This is already in use by various resource monitoring applications/infrastructures. Although SNMP provides remote monitoring, the capability of empowering the user with control over certain aspects of the system lacks. The decentralized architecture tends to involve the user in an interactive, yet simplified mechanism of system/network monitoring. Therefore, this architecture is most suited to situations where there is a requirement of user control over the monitoring processes.

8.2 Possible Enhancements

8.2.1 Integration of more Tools

Many more useful tools can be integrated into EMA due to its scalable architecture. Web100 [1] provides an effective enhanced TCP statistics monitoring framework. EMA clients may interact with web100 servers deployed at various locations to test their TCP performance. This may prove to be very useful for high performance networks, where fault detection requires detailed analysis of the TCP stack.

8.2.2 Performance Data Analysis

Information gathered by EMA can be analyzed for fault detection. Currently, the information is sent to the remote servers where it is persisted in the database. An analysis engine may be developed that performs detailed analysis on the reported information to detect faults in either the host configuration, the TCP parameters collected through the web100 server, or network statistics gathered by the network modules. Based on this analysis, reports may also be generated to facilitate the task of decision making. This may evolve to be an effective network diagnostic infrastructure.

8.2.3 EMA Server Enhancements

EMA Server can serve as a platform for the development of application utilizing various statistical techniques for the development of an inference engine. The data repository provides data in a very useful and easy-to-implement form so that further tools can be made, which can be used for a variety of purposes. Some are listed below:

8.2.3.1 The LAN Perspective

EMA can be a very useful tool for problem diagnosis on various LANs. EMA clients deployed at different nodes server as monitoring clients and EMA server can utilize this monitoring information for problem diagnosis across the LAN. Often faulty cables or improper switch/router configuration causes a lot of disruptions in a network. By building a small analysis engine, EMA can server as the diagnostic tool for common LAN problems.

8.2.3.2 The WAN Monitor

EMA can serve as a useful monitoring tool if deployed over different nodes on a WAN. The network latency and throughput problems are a major concern for those who want to utilize the full potential of high speed WANs. Often this happens due to the presence of just one or two performance bottlenecks along the path. This can be detected by EMA. The detected bottleneck can be upgraded with little overhead cost and provide far better performance without any major changes to the existing systems.

8.2.3.3 EMA for Job Scheduling

EMA can be used by applications on the grid for performing job scheduling and execution based on the information being provided. Since Grid nodes work on a variety of platforms, EMA serves as a suitable candidate for deployment in such scenarios as exactly the same parameters are provided on all EMA supported platforms. Moreover, the network monitoring capability of EMA makes it a far better choice for top-level decision support systems.

8.2.3.4 EMA for detection of Viruses/Worms/Trojan Horses

Certain viruses remain undetected and yet consume resources; Sometimes in form of the end-host's basic computational or storage resources and sometimes in form of network attacks. The graphical trends provided by EMA can serve as a virus diagnostic tool.

8.2.3.5 Monitoring of Nodes in Enterprise Networks

Big enterprises are often interested in the usage of resources provided by them. EMA deployed on clients can serve as a primary monitor of the enterprise's resources.

APPENDICES

Appendix A – Abing

Abing is a tool using the packet pairs dispersion technique to estimate the available Bandwidth/bitrate (unused capacity) for a path in the network. The code is based on the research results described in the paper J.Navratil, R.L. Cottrell "A Practical Approach to Available Bandwidth Estimation" presented in PAM'03 (Passive & Active Measurement Workshop), April 6-8 2003 at La Jolla, California and published in Proceedings issued by San Diego Supercomputing Center UCLA San Diego. pp.1-11.

The method is based on estimating the cross-traffic (or load using a more common terminology) as a basic parameter. The probing packets are sent to the Internet from the abing host with a known separation. We then measure the time between delivery of the adjacent packets in a pair. From this we calculate the utilization of the bottleneck link and other parameters.

The tool was designed to measure available bandwidth in high capacity paths so it can cover all paths with bandwidths between 1-1000 Mbits/s. However, for measurements on high speed links, the appropriate hardware must be used (e.g. a Linux machine with a clock > 1000MHz and a Gbits/s NIC card). The tool is designed to be minimally intrusive and can be run over long durations. It sends only 40 probing packets per measurement.
There are 2 programs which are used during each measurement experiment:

- reflector(server) that is running on remote site (path destination)
- abing (client) which sends probe packet to a reflector,
  receive packets from the reflector and makes the analysis

In the interactive mode, the results are delivered to the terminal immediately after the bunch of probes (20 packet pairs) has traversed the whole path there and back.

5 values are reported for each direction. These are

- ABw estimated Available Bandwidth (instant value)

- Xtr estimated cross-traffic (instant value)
- DBC Dominated Bottleneck Capacity (instant value)
- ABW Average of Available BandWidth calculated via EWMA
- RTT Round Trip Time

The results are presented in the following form:
timestamp, direction flag, ip-address and estimated data as shown in following lines:
>abing -d 132.15.144.226
1075925312 T: 131.15.144.226 ABw-Xtr-DBC: 531.4 277.0 808.3 ABW: 531.4 Mbps RTT: 14.1 ms
1075925312 F: 131.15.144.226 ABw-Xtr-DBC: 371.7 495.9 867.6 ABW: 371.7 Mbps RTT: 14.1 ms

APPENDIX B – Iperf

Iperf is a tool to measure maximum TCP bandwidth, allowing the tuning of various parameters and UDP characteristics. Iperf reports bandwidth, delay jitter, datagram loss. It has two major goals. First, Iperf is a tool for analyzing bandwidth and throughput on a network. Second, Iperf allows the user to tinker with different TCP and UDP parameters to see how they affect network performance. Many users find optimal settings using Iperf and then tune their connections with those values.

Features

- TCP
    - Measure bandwidth
    - Report MSS/MTU size and observed read sizes.
    - Support for TCP window size via socket buffers.
    - Multi-threaded if pthreads or Win32 threads are available. Client and server can have multiple simultaneous connections.
- UDP
    - Client can create UDP streams of specified bandwidth.
    - Measure packet loss
    - Measure delay jitter
    - Multicast capable
    - Multi-threaded if pthreads are available. Client and server can have multiple simultaneous connections. (This doesn't work in Windows.)
- Can run for specified time, rather than a set amount of data to transfer.
- Picks the best units for the size of data being reported.
- Server handles multiple connections, rather than quitting after a single test.
- Print periodic, intermediate bandwidth, jitter, and loss reports at specified intervals.
- Run the server as a daemon
- Run the server as a Windows NT Service

- Use representative streams to test out how link layer compression affects your achievable bandwidth.

Iperf allows the manipulation of most application-accessible network parameters. Commonly manipulated parameters include window size (-w), bandwidth (-b), and time to live (-T).

Iperf sends packets from the client to the server as fast as it can, of course being limited by several factors. The information is sent by default from the client's memory to the server's memory to attempt to eliminate some hardware speed limitations. Note that high bandwidth networks often require multiple streams to max out the bandwidth

Iperf servers can maintain multiple connections at one time. The statistics for each connection will be displayed when it completes and the aggregate statistics will be displayed when all of the connections have finished. In the current version, Iperf servers can limit the number of connections to a user-specified amount.

Appendix C – Java Web Start

Java Web Start software provides a flexible and robust deployment solution for Java technology-based applications based on the Java Community Process program (JCP). The technology is being developed through the JCP program as JSR-56: The Java Network Launching Protocol & API (JNLP), which provides a browser-independent architecture for deploying Java 2 technology-based applications to the client desktop.

Java Web Start technology works with any browser and any Web server. Each application developed for use with the Java Web Start software specifies which version of the Java 2 platform it requires, e.g., version 1.2 or 1.3, and each application runs on a dedicated Java Virtual Machine (JVM).

A main feature of the Java Network Launching Protocol and API technology is the ability to automatically download and install Java Runtime Environments onto the users machine.

For example, an application might depend APIs in Sun's Java Runtime Environment 1.3.0 (or later). When a user first accesses this application, the Java Web Start software will download all the needed files for the application, as well as download the Java Runtime Environment (JRE) if the requested version is not available locally. The ability to automatically download a JRE is one of the key features to ensure robust deployments. It ensures that the JRE that your application is tested on will be available on the user's machine, as well as make it possible to seamlessly upgrade to improved versions of the Java 2 platform over time.

When to use Java Web Start Technology:

Given the nature of many productivity tools and traditional client-side applications, providing an HTML-based front-end to the application or service is not adequate.

For example, Web-based e-mail has been widely successful. It is a convenient tool when the volume of e-mail is relative low. Because most corporate users of e-mail and many individuals receive high volumes of e-mail, typically dozens a day, using an HTML-based interface would prove cumbersome and ineffective.

With Java Web Start technology, which works with virtually all Web servers, the application service providers (ASP), either internally to the company or externally on the Web, can easily supply a full-featured application to users. Java Web Start technology is an ideal companion to HTML-based clients. A service can provide a simple and easily accessible interface using HTML, while also providing a rich experience for power users using Java Web Start technology. The ease-of-use for users is virtually identical -- except for the first-time activation cost -- and the management and server side requirements for both solutions are the same.

Features of Java Web Start Technology:

Applications written on the Java 2 platform for deployment with Java Web Start are always up-to-date and available. Every application runs in a dedicated Java Runtime Environment (JRE), independent of a specific browser or computer platform technology. Java Web Start supports:

- Multiple JREs
- Code-signing
- Sandboxing
- Versioning and incremental updates
- Desktop integration
- Offline operation
- Automatic installation of JREs and optional packages Application launcher

Java Web Start technology provides a rich set of features that give easy access to the latest versions of applications for the end-user, easy management and deployment of applications for the IT department, and easy development for the application vendors.

Appendix D – Load Average

UNIX Load Average Part 1 How it works by Dr. Neil Gunther, Performance Dynamics Company
http://www.teamquest.com/resources/gunther/ldavg1.shtml

1 UNIX™ Commands

Actually, *load average* is not a UNIX™ command in the conventional sense. Rather it's an embedded metric that appears in the output of other UNIX™ commands like `uptime` and `procinfo`. These commands are commonly used by UNIX™ sys admin's to observe system resource consumption. Let's look at some of them in more detail.

1.1 Classic Output

The generic ASCII textual format appears in a variety of UNIX™ shell commands. Here are some common examples.

- uptime
- procinfo
- w
- top

2 So What Is It?

So, exactly what is this thing called *load average* that is reported by all these various commands? Let's look at the official UNIX™ documentation.

2.1 The man Page

In the man page for `uptime`, for example, and see if we can learn more that way.

```
...
DESCRIPTION
       uptime  gives a one line display of the following informa-
       tion.  The current time, how long the system has been run-
       ning, how many users are currently logged on, and the sys-
       tem load averages for the past 1, 5, and 15 minutes.
...
```

So, that explains the three metrics. They are the ``... load averages for the past 1, 5, and 15 minutes."

3 Performance Experiments

The experiments described in this section involved running some workloads in background on single-CPU Linux box. There were two phases in the test which has a duration of 1 hour:

- CPU was pegged for 2100 seconds and then the processes were killed.
- CPU was quiescent for the remaining 1500 seconds.

A Perl script sampled the load average every 5 minutes using the `uptime` command. Here are the details.

3.1 Test Load

Two hot-loops were fired up as background tasks on a single CPU Linux box. There were two phases in the test:

1. The CPU is pegged by these tasks for 2,100 seconds.
2. The CPU is (relatively) quiescent for the remaining 1,500 seconds.

The 1-minute average reaches a value of 2 around 300 seconds into the test. The 5-minute average reaches 2 around 1,200 seconds into the test and the 15-minute average would reach 2 at around 3,600 seconds but the processes are killed after 35 minutes (i.e., 2,100 seconds).

3.2 Process Sampling

As the authors [BC01] explain about the Linux kernel, because both of our test processes are CPU-bound they will be in a `TASK_RUNNING` state. This means they are either:

- *running* i.e., currently executing on the CPU
- *runnable* i.e., waiting in the run_queue for the CPU

The Linux kernel also checks to see if there are any tasks in a short-term sleep state called `TASK_UNINTERRUPTIBLE`. If there are, they are also included in the load average

sample. There were none in our test load. The following source fragment reveals more details about how this is done.

```
600  * Nr of active tasks - counted in fixed-point numbers
601  */
602 static unsigned long count_active_tasks(void)
603 {
604         struct task_struct *p;
605         unsigned long nr = 0;
606
607         read_lock(&tasklist_lock);
608         for_each_task(p) {
609                 if ((p->state == TASK_RUNNING ||
610                      (p->state & TASK_UNINTERRUPTIBLE)))
611                         nr += FIXED_1;
612         }
613         read_unlock(&tasklist_lock);
614         return nr;
615 }
```

So, `uptime` is sampled every 5 seconds which is the linux kernel's intrinsic time base for updating the load average calculations.

3.3 Test Results

Although the workload starts up instantaneously and is abruptly stopped later at 2100 seconds, the load average values have to catch up with the instantaneous state. The 1-minute samples track the most quickly while the 15-minute samples lag the furthest.

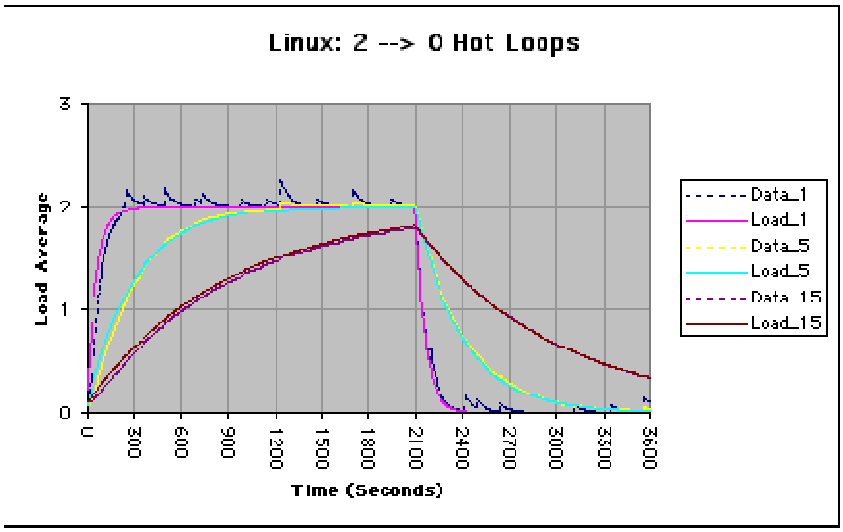

Figure 2: Linux load average test results.

For comparison, here's how it looks for a single hot-loop running on a single-CPU Solaris system.

4 Kernel Magic

Now let's go inside the Linux kernel and see what it is doing to generate these load average numbers.

```
unsigned long avenrun[3];
624
625 static inline void calc_load(unsigned long ticks)
626 {
627         unsigned long active_tasks; /* fixed-point */
628         static int count = LOAD_FREQ;
629
630         count -= ticks;
631         if (count < 0) {
632                 count += LOAD_FREQ;
633                 active_tasks = count_active_tasks();
634                 CALC_LOAD(avenrun[0], EXP_1, active_tasks);
635                 CALC_LOAD(avenrun[1], EXP_5, active_tasks);
636                 CALC_LOAD(avenrun[2], EXP_15, active_tasks);
637         }
638 }
```

4.1 Magic Numbers

The function `CALC_LOAD` is a macro defined in sched.h

```
58 extern unsigned long avenrun[];         /* Load averages */
59
60 #define FSHIFT          11              /* nr of bits of precision
*/
61 #define FIXED_1         (1<<FSHIFT)     /* 1.0 as fixed-point */
62 #define LOAD_FREQ       (5*HZ)          /* 5 sec intervals */
63 #define EXP_1           1884            /* 1/exp(5sec/1min) as
fixed-point */
64 #define EXP_5           2014            /* 1/exp(5sec/5min) */
65 #define EXP_15          2037            /* 1/exp(5sec/15min) */
66
67 #define CALC_LOAD(load,exp,n) \
68         load *= exp; \
69         load += n*(FIXED_1-exp); \
70         load >>= FSHIFT;
```

A notable curiosity is the appearance of those magic numbers: 1884, 2014, 2037. What do they mean? If we look at the preamble to the code we learn,

```
/*
49  * These are the constant used to fake the fixed-point load-average
50  * counting. Some notes:
51  *  - 11 bit fractions expand to 22 bits by the multiplies: this
gives
52  *    a load-average precision of 10 bits integer + 11 bits
```

```
fractional
53  *  - if you want to count load-averages more often, you need more
54  *    precision, or rounding will get you. With 2-second counting
freq,
55  *    the EXP_n values would be 1981, 2034 and 2043 if still using
only
56  *    11 bit fractions.
57  */
```

These magic numbers are a result of using a fixed-point (rather than a floating-point) representation.

Using the 1 minute sampling as an example, the conversion of exp(5/60) into base-2 with 11 bits of precision occurs like this:

$$e^{5/60} \rightarrow \frac{e^{5/60}}{2^{11}} \qquad (1)$$

But EXP_M represents the inverse function exp(-5/60). Therefore, we can calculate these magic numbers directly from the formula,

$$\text{EXP\_M} = \frac{2^{11}}{2^{5 \log_2(e) / 60M}} \qquad (2)$$

where M = 1 for 1 minute sampling. Table 1 summarizes some relevant results.

T	EXP_T	Rounded
5/60	1884.25	1884
5/300	2014.15	2014
5/900	2036.65	2037
2/60	1980.86	1981
2/300	2034.39	2034
2/900	2043.45	2043

Table 1: Load Average magic numbers.

These numbers are in complete agreement with those mentioned in the kernel comments above. The fixed-point representation is used presumably for efficiency reasons since these calculations are performed in kernel space rather than user space.

One question still remains, however. Where do the ratios like exp(5/60) come from?

4.2 Magic Revealed

Taking the 1-minute average as the example, `CALC_LOAD` is identical to the mathematical expression:

$$load(t) = load(t\text{-}1)\ e^{-5/60} + n\ (1 - e^{-5/60}) \qquad (3)$$

If we consider the case n = 0, eqn.(3) becomes simply:

$$load(t) = load(t\text{-}1)\ e^{-5/60} \qquad (4)$$

If we iterate eqn.(4), between $t = t_0$ and $t = T$ we get:

$$load(t_T) = load(t_0)\ e^{-5t/60} \qquad (5)$$

which is pure exponential decay, just as we see in Fig. 2 for times between $t_0 = 2100$ and $t_T = 3600$.

Conversely, when n = 2 as it was in our experiments, the load average is dominated by the second term such that:

$$load(t_T) = 2\ load(t_0)\ (1 - e^{-5t/60}) \qquad (6)$$

`which is a monotonically increasing function just like that in Fig. 2 between $t_0 = 0$ and $t_T = 2100$.

5 Summary

1. The ``load'' is not the utilization but the total queue length.
2. They are point samples of three different time series.
3. They are exponentially-damped moving averages.
4. They are in the wrong order to represent trend information.

Appendix E – MonALISA

http://monalisa.cacr.caltech.edu/

The MonALISA framework provides a distributed monitoring service system using JINI/JAVA and WSDL/SOAP technologies. Each MonALISA server acts as a dynamic service system and provides 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.

The MonALISA architecture provides

- Mechanism to dynamically discover all the "Farm Units" used by a group or community
- Remote event notification for changes in the any system
- SNMP support and interfaces with other tools: Ganglia, MRTG, LSF, PBS, user defined scripts
- Flexible mechanism to add additional modules or interfaces to other tools
- Secure mechanism for dynamic configuration of farms / network elements and the collected parameters
- Access to single farm values and all the details for each node
- Network parameters, connectivity values and traffic information
- Selected real time data for any subscribed listeners
- Selected historical data using a flexible persistent mechanism based on JDBC
- Active filters to process the data and provided dedicated / customized information to other services
- Mobile agents to control different activities in the system
- Alarm triggers
- Dynamic proxies (or WSDL)so that clients can access the information in a flexible way
- Configurable GUIs to present from real-time global views of multiple farms to the evolution in time of any single parameter

- Authentication and secure GUI connection to configure and administrate a monitoring service
- Global monitoring repositories for a community
- Access to the monitoring information from mobile phones using WAP

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 a 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 that are collected by this module.

Creating a new module means writing a class that extends the
`lia.Monitor.monitor.cmdExec` class and implements
`lia.Monitor.monitor.MonitoringModule`

interface.

The `doProcess` is actuly the function that collects and returns the results. Usually the return type is a Vector of `lia.Monitor.monitor.Result` objects. It can also be a simple `Result` object.

The `init` function initializes the useful information for the module, like the cluster that contains the monitoring nodes, the farm and the command line parameters for this module. This function is the first called when the farm loads the module.

The `isRepetitive` function tells if the module has to collect results only once or repetitively. The return values is the `isRepetitive` module boolean variable. If true, then the module is called from time to time. The repetitive time is specified in the <farm> `.conf` file. If not there, then the default repetitive call time is 30s.

The rest of functions returns different module information.

This interface has the following structure:

```
package lia.Monitor.monitor;

public interface MonitoringModule extends
lia.util.DynamicThreadPoll.SchJobInt {

    public  MonModuleInfo init( MNode node, String
args ) ;

        public String[] ResTypes() ;

        public String   getOsName();

        public Object   doProcess() throws Exception
;

        public MNode getNode();

        public String getClusterName();

        public String getFarmName();

        public boolean isRepetitive() ;

        public String getTaskName();

        public MonModuleInfo getInfo();


    }
```

Bibliography

1. Olivier Martin, “Gigabits, the Grid and the Guinness Book of Records”, *http://www.cerncourier.com/main/article/44/2/14*
2. Matt Mathis John Heffner Raghu Reddy, “Web100: Extended TCP Instrumentation for research, education and diagnoses”, *ACM SIGCOMM CCR*, July 2003
3. E2E pipes, *http://e2epi.internet2.edu/E2EpiPEs/index.html*
4. Foster, I., Kesselman, C. and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 200-222. 2001.
5. Catlett, C. In Search of Gigabit Applications. *IEEE Communications Magazine* (April). 42-51. 1992.
6. Catlett, C. and Smarr, L. Metacomputing. *Communications of the ACM*, 44--52. 1992.
7. Foster, I. The Grid: A New Infrastructure for 21st Century Science. *Physics Today*, 42-47. 2002.
8. Pen Computing, *http://www.upenn.edu/computing/i2/*
9. Internet2, *http://www.internet2.edu/e2epi/e2epi/papers/End-to-End-Perf-Design-Paper.pdf*
10. Eric L. Boyd, George Brett, Russ Hobby, Jooyong Jun, Clyde Shih, Ramanuja Vedantham, and Matt Zekauskas, E2E piPEline: End-to-End Performance Initiative Performance Environment System Architecture ,Version: 1.1, July 26, 2002
11. Shawn McKee, Pipefitters Meeting,Internet2 Spring Meeting 8 April, 2003
12. Why use open source Technology, *http://www.celestialgraphics.com/articles_open_source_print.html*
13. MSDN, *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/perfmon/base/performance_monitoring_architecture.asp*
14. MSDN, Chapter 5 An overview of performance monitoring, *http://www.microsoft.com/resources/documentation/windows/2000/server/reskit/en-us/serverop/part2/sopch05.mspx*
15. MSDN, Performance Objects and Counters, *http://msdn.microsoft.com/library/en-us/perfmon/base/performance_objects_and_counters.asp*
16. Linux Man Pages
17. Dr. Neil Gunther, UNIX Load Average Part 1 How it works, *Performance Dynamics Company*, 2001
18. Tom M. Mitchell, *Machine Learning*, McGraw Hill, 1997
19. Monitoring Agents using a Large Integrated Services Architecture, *http://monalisa.cacr.caltech.edu*,2003