

To the Graduate Council:

I am submitting herewith a thesis written by Krishna Kanth Inavolu entitled “STATMOND: A Peer-To-Peer Status And Performance Monitor For Dynamic Resource Allocation On Parallel Computers”. I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Dr. John D. Birdwell

Major Professor

We have read this thesis
and recommend its acceptance:

Dr. Tse-Wei Wang

Dr. Hairong Qi

Dr. John Chiasson

Accepted for the Council:

Anne Mayhew

Vice Chancellor
and Dean of Graduate Studies

(Original signatures are on file with official student records.)

**STATMOND: A PEER-TO-PEER STATUS AND
PERFORMANCE MONITOR FOR DYNAMIC
RESOURCE ALLOCATION ON PARALLEL
COMPUTERS**

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Krishna Kanth Inavolu

May 2004

Copyright © 2004 by Krishna Kanth Inavolu

All rights reserved

*To my parents, Maheswari and Seshachala Rao (Inavolu), who have blessed my
life with their love, support, and sacrifices*

Acknowledgements

This work was supported by the U.S. Department of Justice, Federal Bureau of Investigation under contract J-FBI-98-083. The views and conclusions contained in this thesis are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Government.

I gratefully acknowledge the support and presence in this work of many people without whom I would have never completed the program. I am grateful to my parents, for their undying faith and love; my major advisor Dr. J. Douglas Birdwell for his guidance, advice, insight, and financial support; the members of my thesis committee: Dr. Tsewei Wang, Dr. John Chiasson, and Dr. Hairong Qi, for their valuable input and feedback.

I am also indebted to the staff and student members of the Laboratory for Information Technologies (LIT) especially, Dr. Mark Rader, Zhong Tang and Zeqian Shen whose help, stimulating suggestions and encouragement helped me

during the entire time of research and writing of this thesis.

Few people will understand a word of this thesis, yet it would not exist without the club. In particular, Hafizur Rahman, my roomie Sivakali Dasari, Kiran Sagi, and Teja Kuruganti have shared my good times at UT, Knoxville. Other friends whose encouragement and support have remained constant in spite of the distance are Vikram Palakodety, Srinivas Jillela, Ashok Devata, and Vamsi Ivaturi.

Abstract

This thesis presents a decentralized tool STATMOND - to monitor the status of a peer-to-peer network. STATMOND provides an accurate measurement scheme for parameters such as CPU load and memory utilization on Linux clusters. The services of STATMOND are ubiquitous in that each computer measures and forwards its data over the network and also maintains the data of other nodes in memory. The data are periodically updated, and users on any node can ‘see’ the status and performance of the network based on these parameters. This thesis describes the problems confronting cluster computing, the necessity of monitoring tools and how STATMOND can be a step towards better allocation of resources for dynamic computing.

Contents

1	Introduction	1
1.1	Introduction	1
1.2	Motivation	3
1.3	The Specific Problem	5
1.4	An Overview of the Solution	7
1.5	Summary	8
2	Background	9
2.1	Introduction	9
2.2	Forensic Science And Solving Crime	10
2.3	The Combined DNA Index System	12
2.3.1	The CODIS Framework	13
2.4	Parallel Machine at LIT and the Parallel Search System	14
2.5	Dynamic Resource Allocation on a Parallel Computer	16

2.5.1	Nodes, Processors and their Constraints	18
2.5.2	Memory and its Constraints	20
2.5.3	The Network and its Constraints	21
2.5.4	Goal of DRA	22
2.6	STATMOND and DRA	24
2.7	Summary	25
3	Survey of Related Tools	26
3.1	Introduction	26
3.2	Monitoring Tools Developed in Academia	27
3.3	Monitoring Tools Developed in the Industry	33
3.4	Other Monitoring Tools	34
3.5	Summary	37
4	STATMOND Design Overview	39
4.1	Introduction	39
4.2	STATMOND Process	39
4.3	STATMOND Kernel Instrumentation	42
4.4	STATMOND Data	45
4.5	Data Transfer Between STATMOND Processes	48
4.6	The STATMOND-Client	50

4.7	Summary	53
5	Implementation of STATMOND	55
5.1	Introduction	55
5.2	Detailed Design of STATMOND	56
5.2.1	Data Handling in STATMOND	60
5.2.2	Multithreading in STATMOND	62
5.2.3	Multicasting through STATMOND	67
5.2.4	STATMOND's Output Representation	70
5.2.5	Custom Filters	76
5.3	Detailed Design of the Client	79
5.4	Summary	83
6	Analysis and Future Work	84
6.1	Introduction	84
6.2	Review of Work	84
6.3	Analysis of STATMOND	87
6.3.1	Non-Intrusiveness	87
6.3.2	Fault-tolerance	92
6.3.3	Management of Monitored Data	92
6.3.4	Use of Standard Data Formats	93

6.4	Comparison of STATMOND and Other Tools	93
6.5	Future Work	95
6.5.1	Improve STATMOND's Scalability	95
6.5.2	Provide a Front-End	97
6.5.3	Extend the Filter Framework	99
6.6	Conclusions	99
6.7	Summary	99
	Bibliography	101
	Vita	110

List of Tables

2.1	Nodes, Processors, and their Constraints	19
2.2	Memory and its Constraints	21
2.3	Network and its Constraints	23

List of Figures

2.1	Parallel Database System[37]	14
3.1	The Structure of Network Weather Service[57]	28
3.2	NWS Servers Running on Three Monitored Hosts[57]	29
4.1	STATMOND Framework with 6 Nodes	40
4.2	General STATMOND Table Structure	47
4.3	Unicast Loads the Sender	49
4.4	Sending Packet to a Multicast Group	51
4.5	Multicast Enables the Sender to Send Just Once	51
4.6	Client-STATMOND Interaction	52
5.1	Incorrect Command Line Execution of STATMOND	57
5.2	A Correct Command Line Execution of STATMOND	57
5.3	STATMOND's PID	58
5.4	Internal Software Structure of STATMOND	64

5.5	The Probe Thread	64
5.6	The Send Thread	65
5.7	The Send Thread	66
5.8	The Client Thread	67
5.9	Snapshot of XML Data File Contents	72
5.10	XML Data File Contents of a Uni-processor Node	73
5.11	XML Data File Detailed Contents of a Uni-processor Node	74
5.12	XML Data File Detailed Contents of a Multi-processor Node . . .	75
5.13	Custom Filter - STATMOND Interaction	76
5.14	Processes on Node10 (192.168.1.10)	77
5.15	Command Line Execution of Custom Filter	78
5.16	XML Data File of Custom Filter	78
5.17	XML Data File for Client	82
6.1	%CPU usage on NODE01, No output stored	89
6.2	%STATMOND NODE 01 CPU usage, Monitoring Interval: 5 secs	89
6.3	%STATMOND NODE 01 CPU usage, Monitoring Interval: 15 secs	90
6.4	%STATMOND NODE 00 CPU usage, Monitoring Interval: 5 secs	91
6.5	%STATMOND NODE 00 CPU usage, Monitoring Interval: 15 secs	91
6.6	Scalability Problem: A Much Better Solution	97

Chapter 1

Introduction

1.1 Introduction

Parallel processing applications use more than one central processing unit (CPU) to execute a program at speeds in excess of what can be achieved by a conventional computer. Some problems in science and engineering are both memory and computationally intensive, such as linear algebra routines on very large matrices, finite element computing, and searches in large databases. The time required for one processor (CPU) to execute these applications can be unacceptably long. Such applications can require both gigabytes of memory and gigaflops (1 FLOPS = 1 floating point operation/second, where the floating point operation is a multiply and add or equivalent) of performance. Often this is more than what can

reasonably be expected to be available on a single processor machine. The main objective of any computation is to obtain a sufficiently accurate solution within an acceptable amount of time. One way to solve such complex problems is to divide the computational work into discrete sections called tasks or jobs and assign them to several CPUs, so that the tasks are performed simultaneously. Parallel processing thus involves two or more CPUs interconnected through any communication medium such as a network, with each CPU executing a task.

Parallel processing, also known as parallel computing, can be implemented in several ways. These include cluster computing and meta-computing (grid computing). Instead of spending millions of dollars on specialized supercomputers such as a Cray X1, or a HP/Convex Exemplar SPP-1200, it is often more cost-effective to build and maintain a cluster of computers that lets users plug into computing power on a network very easily. Grid computing deals with a much larger cluster of computers, that offers teraflops of performance and spans multiple geographic and administrative domains. A cluster of computers/workstations, popularly known as a Network of Workstations (NOW)[59], uses commodity microprocessors and high-speed local area network (LAN) interconnects such as Fast or Gigabit Ethernet. It acts as a single system and can have performance competitive with supercomputers that are more monolithic. Clusters also scale well, accommodating a large number of computers that can be added in an incremental

fashion.

A cluster can be built with common LAN resources, such as individual computers, high-speed (several GHz) processors, memory (several GB), disks, I/O cards, network interface cards, cables, operating systems and other utility software (such as mathematical libraries). To meet application requirements, several computers in a cluster can be connected to two or more LAN architectures. Such computers, or nodes, associate themselves with different LANs through distinct interfaces identified by unique IP addresses. One or more of these resources may fail, and it is desired that the program tolerate such failures. This presents the parallel architecture designers with both an opportunity and a challenge. The opportunity is that the computation may be designed to survive despite the failure of one or more resources. The challenge is to be able to monitor such a system, detect the failures of individual resources, and design methods to adapt or reconfigure the environment and parallel application to minimize the effects of failures.

1.2 Motivation

Dynamic resource allocation (DRA) in a cluster of workstations is one of the most important problems that must be solved for efficient utilization of all resources (nodes/parallel computers/hosts, databases, network links, storage, I/O devices) in a cluster. The main objective of a DRA method is to allocate and map

workloads to each resource in proportion to its capability. A well-known similar problem is how to manage, schedule, distribute, and delegate activities among people working on a project in a company. The DRA method must adjust to changes in its set of available information. Other objectives include:

1. Workload distribution over a set of processors and resources.
2. Intelligent planning of the distribution of workload, so that all the computers are busy and at the same time not overloaded.
3. Assignment and adjustment of task priority.
4. Efficient and rapid completion of assigned work.
5. Maximization of task throughput on the machines.

A resource scheduler must use host capacity information to achieve these objectives. However, this information cannot be exact or complete. First, only a finite (typically small) number of statistics can be available for each host. Second, these statistics are measured infrequently (relative to the performance of the hosts). This ensures that the measurement processes consume a small fraction of the parallel machine's resources. Third, once statistics are measured they must be transmitted to other node(s) and made available to the resource scheduling method. This process requires time and creates a randomly varying time delay inherent to the measurement, transmission, and receipt of the information.

1.3 The Specific Problem

This thesis addresses a small but important step that may aid the DRA method. The specific objective of this thesis is to develop a tool that executes as a service daemon to monitor, gather, and distribute the capacity and utilization statistics for DRA. As a daemon or background process, the tool should collect data from the kernel data structures at specified discrete times. This information has to be exchanged with all other nodes, and requests should be serviced to provide information to other software applications. The daemon aims to provide status information, that includes measures of CPU activity, memory and network utilization for every host on the network. Specifically, the data that need to be collected includes:

1. The number of CPUs on a node.
2. The clock speeds of the respective CPUs on a node.
3. The total RAM used by a node.
4. The total amount of memory available on a node.
5. The memory shared by various applications on a node.
6. The memory available in the buffer.
7. The memory available in cache.

8. The number of jiffies (a jiffy is defined as 1/100th of a second) the node has been in user mode.
9. The number of jiffies the node executes tasks is in low priority mode.
10. The number of jiffies the node executes tasks in system mode.
11. The number of jiffies the node is idle.
12. The number of seconds for which the node has been up and was in idle state.
13. The utilization of the processor/s in the last 1, 5, and 10 minutes, and
14. The number of packets sent and received by a node on the network.

A DRA design effort can use this information, along with additional software process information, to design methods to balance the distribution of databases or other applications across the resources of the parallel machine. This distribution can occur under dynamic conditions that include node failures, node maintenance, and hardware/software upgrades. The tool's design is unique: it provides local estimates of properties of all participating nodes on all nodes of the parallel computing environment. The tool is peer-to-peer since each node has the same capabilities and either node can initiate or join a communication session. Other models with which it might be contrasted include the client/server model and the master/slave model. The nodes can use the ethernet connection to exchange

information with each other directly or through a mediating server. The information provided by this tool can also be used by the system administrators and developers to instrument system resources such as CPUs, memory, network, and selected applications.

1.4 An Overview of the Solution

STATMOND (short for Status Monitor Daemon), is designed and implemented to gather, store, transmit, receive, and report the environmental, kernel, system configuration and process data for all the resources (nodes) on a parallel machine. The key features of STATMOND are:

1. It provides accurate and timely statistics.
2. It maintains aggregate data on every node on the network.
3. It is non-intrusive; it uses a small fraction of CPU and network resources.
4. It is available at all times; it does not execute and die.
5. It is ubiquitous; it is locally available at all nodes, and it enables local access to all performance and environmental data on all nodes.

STATMOND has been tested on a 24-node parallel machine and may subsequently be used in the design of a DRA method. The statistics reported by STATMOND

are currently used by system administrators and developers who are responsible for the system.

1.5 Summary

The problem statement and a broad overview of the thesis have been presented in this chapter. The "status and performance measurement of parallel computers" problem has been outlined along with the aim and objectives of this thesis. The next chapter will further describe the problem's setting, its background, and who will benefit from the solution.

Chapter 2

Background

2.1 Introduction

Every cluster is designed to perform a certain set of computations and tasks effectively. The cluster at the Laboratory for Information Technologies (LIT) at the University of Tennessee, Knoxville (UTK) uses the computational power of several (24) nodes to run tasks that store, retrieve and maintain DNA information. This chapter describes, in brief, the setup of the cluster, the database framework that stores and utilizes the DNA data, and the DRA method that uses the resources to achieve desired performance. This overview will provide an insight to the use of STATMOND in the above context.

2.2 Forensic Science And Solving Crime

Everyday, many crimes are committed in society. Forensic science has become an essential tool to solve these crimes. The word "forensic" in the past meant anything relating to a law court. But today, it refers to using science to solve crime.

What exactly is the role of a forensic scientist? At the scene of the crime, the investigators will thoroughly examine the area. They take detailed notes and photographs, look for fingerprints, marks of tools and weapons, marks from shoes and palm prints, cloth or material fibers, glass and paint fragments, saliva, hair and body fluids.

The presence of blood, saliva or hair, however, does not mean it is from a human, or even that it had anything to do with a crime. Occasionally, people do cut themselves, which can leave a tiny drop of blood on the carpet. Similarly an uncleaned carpet may have hair that are dropped from a comb.

Hence, it is not the forensic experts' job to confirm a crime. Forensic science is similar to any form of science and requires an open mind. The crime scene investigator or the lab scientist works to uncover facts that can be used as evidence. Very often, these facts may serve to rule out many of the suspects.

Autopsy, fingerprinting, and DNA fingerprinting are tools used by forensic pathologists to study the cause and nature of the crime. DNA identification is

an emerging technology that relies on the genetic information of each person. Everyone's DNA in certain regions, except from identical twins is different. DNA can be extracted from any body fluid including blood, saliva, sweat, nasal mucus, and semen and other fragments such as hair roots, skin flakes, or flesh. The extracted DNA can be typed for human identification.

Forensic scientists look at a sub-set of the DNA from body fluid sample and perform tests to determine the parentage and pedigree, which is known as DNA profiling. DNA profiles (obtained as a result of DNA profiling) are a very powerful means of determining whether two or more samples may or may not have come from the same person. If DNA profiles do not match, it is highly probable that they came from different people.

Human DNA is classified as either nuclear or mitochondrial DNA. Nuclear DNA originates in the nucleus of each cell, while mitochondrial DNA originates in the mitochondria outside the nucleus of the cell. Mitochondrial DNA are strictly inherited from the maternal side of a person.

Nuclear DNA is the same in every cell of the body, and stays the same throughout life. Therefore, DNA profiles taken at different times and places can be compared in order to determine whether or not they come from the same person. DNA profiling is used to compare different DNA samples, and to assist in the determination of whether or not they could be from the same person. Various

probabilities of DNA matching can be carried out to assess the likelihood of two DNA profiles being the same if they are from different sources.

2.3 The Combined DNA Index System

The Combined DNA Index System or CODIS [11], established by the Federal Bureau of Investigation (FBI), is a database software that can store (maintain), track, and search DNA profiles. These profiles can originate from offenders convicted of serious crimes that include felony and sexual assault, from forensic investigations of crime scenes, and other related sources such as DNA data for missing persons. The CODIS software is used in local, state and national repositories of DNA profiles created from convicted offender samples and case evidence (such as saliva, blood, chewing gum, hair, cigarette butts, etc.) left behind by individuals at a crime scene. Law enforcement agencies use CODIS to match DNA profiles in the database with those obtained from DNA samples collected from various crime scenes or from suspects in custody. In short, CODIS makes effective use of forensic science and computer technology to trace criminals and solve crimes.

2.3.1 The CODIS Framework

The CODIS software currently is used at three geographic levels:

1. National DNA Index System (NDIS)
2. State DNA Index System (SDIS)
3. Local DNA Index System (LDIS)

At the national level, CODIS stores DNA profiles from crime scenes, convicted offenders, and from people who have submitted a blood sample in compliance with the existing law. NDIS is maintained by the Federal Bureau of Investigation and is used by federal, state, and law enforcement agencies. This was formalized by the DNA Identification Act of 1994. Through the NDIS, DNA profiles are exchanged and interstate comparisons are performed.

Each member state has a respective SDIS database, and may have one or more LDIS databases. Each LDIS database is comprised of at least three indices: casework, convicted offender, and population. To search for matching DNA profiles, law enforcement agencies use the casework and convicted offender indexes. The crime scene evidence is searched against the DNA profiles maintained within the convicted offender indexes. This enables the forensic science agencies to link unsolved and unrelated cases and also to identify unknown suspect cases.

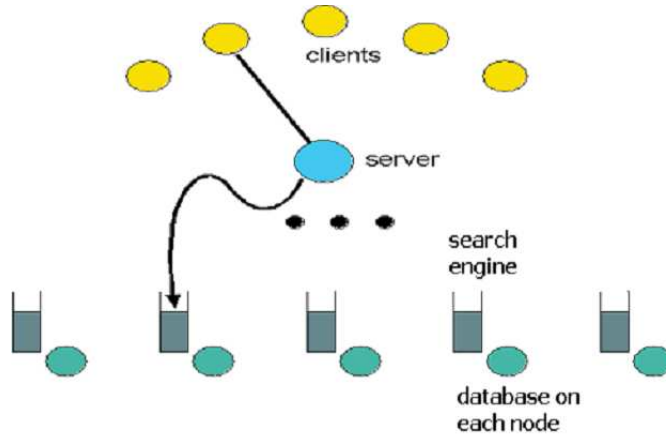


Figure 2.1: Parallel Database System[37]

2.4 Parallel Machine at LIT and the Parallel Search System

A parallel computer system was constructed at LIT for architecture, software development tools, and applications research. The parallel system is a collection of networked computers that can support parallel databases and collections of databases. Primarily built for the FBI Laboratory, this system implements a parallel DNA profile database [39][40]. The layout of the parallel database is shown in Figure 2.1[37].

The structure of the parallel database is hierarchical. It consists of a root node that communicates with k groups of computer networks. Each group is assigned a portion of the database and is composed of one or more computer

nodes based on their hardware configurations, that is, processor speed, memory size, and other features that contribute to the computational speed of the node. All of the nodes within a group contain identical data , and the nodes share the computational burden of database searches involving that group. Different groups contain different DNA profile data, but one or more groups can store redundant copies of portions of the total data population to ensure data integrity in the event of dynamic node failures. In the LIT parallel computer cluster each node has one or two AMD Athlon processors. The single processor machines use a 1.3 GHz processor and 768 MB RAM (Random Access Memory), and the dual processor machines use two 1.6 GHz processors and 1GB RAM. Red Hat Linux is the operating system. The TCP/IP and UDP protocol suite on both Gigabit and Fast Ethernet is used for communications between the nodes.

The LIT parallel system has 2 root nodes, 24 nodes, 40 processors, and 3.5 Terabytes of disk storage. The database implements a tree-structured index that specifies a sequence of tests on a DNA target profile to determine where it should be stored (or where matching profiles would be stored). Data records or DNA profiles are stored as sets of clusters. The database responds to client queries for search, storage, deletion and retrieval of data. The root node creates a query based on the request from the client. The root along with all the nodes maintains a list of the processing capacity of each processor, available memory, queue length

and the number of tasks on each processor. Based on this information a set of tests are defined by the root node, and each node is associated with a different test. These tests are processed by each node in parallel. The result of a test is the response to the query and each node routes it to the root node. The root evaluates the responses for the query and sends the appropriate result to the client.

This parallel data processing architecture is the basis for the parallel search system at LIT. In short, the parallel database server is a tree of cooperating hosts organized in groups. Cooperation is necessary as the search requests are not created only by the clients but by any node that executes a search thread. Pending search requests, placed on queues associated with each node may be moved among the processing nodes of a group to achieve a balance of the load.

2.5 Dynamic Resource Allocation on a Parallel Computer

The objective of dynamic resource allocation (DRA) is to allocate available system resources to tasks under dynamic conditions caused by the completion and insertion of new tasks, changing utilization patterns of system resources, node failures, system and software maintenance, database growth, and hardware/software upgrades, in a manner that maintains suitable performance. The entire search

system discussed in the previous section consists of a space server (root node) and a group of nodes connected through a switch. The resources of this system include:

1. Processors on each node (A node can have 1 or 2 processors).
2. Memory on each node.
3. Bandwidth available to two communicating nodes across the network.

Issues that need to be taken into consideration in assigning resources to the DNA profile database search include:

1. The initial size of the entire database (The initial quality of DNA profiles to be allocated to each node depends on the available RAM and processor speed of the respective node).
2. The rate of growth of the database.
3. The number of node-failures that can be handled.
4. The size of database that will be redistributed in case of failure of a particular node.
5. The size of the database that will be reassigned when new nodes or resources are introduced.

6. The role of the hot spare (back up computer).
7. Overlapping or non-overlapping databases

In addition, maintenance of these nodes is required from time to time, and there should be a procedure that accomplishes this without severely impacting performance, that is the service time of the requests.

The resources of the network are subject to constraints. In order to maintain suitable performance of the system, the relationships between the resources and their constraints are analyzed to determine the amount of resources required and their allocation [47][45].

2.5.1 Nodes, Processors and their Constraints

This section introduces the parameters and the variables that characterize the number of nodes, the size of the database, and the rate of incoming search and store requests. The entire search system needs to be designed to achieve faster search rates. However, the database grows with time. So, the number of records that need to be searched to find a match increases the average search time over time. One possible solution is to add more nodes, as the database size grows so as to maintain a constant search time. To analyze several possible alternative solutions, the parameters and variables, and their units, are defined in Table 2.1.

Table 2.1: Nodes, Processors, and their Constraints

Symbol	Description
$\mathbf{n}$	The number of nodes on the network.
$\mathbf{DB(t)}$	The total database of profiles (Gbytes) that must be searched at time t .
$\mathbf{Q_i(t)}$	The number of searches per second of the i^{th} node at time t .
$\mathbf{S_i(t)}$	The number of new profile stores per second of the i^{th} node at time t .
$\mathbf{DB_i(t)}$	The database size on the i^{th} node (Gbytes) at time t . Constraint: $\sum_{i=1}^n DB_i(t) = DB(t)$
$\mathbf{CPU_i}$	The processor capacity of the i^{th} node (GHz) With $\alpha \ln(\text{Gbytes})$ -searches deduced empirically, a model of the search rate is: $Q_i(t) = \alpha \frac{CPU_i}{\ln(DB_i(t))}$
$\mathbf{q(t)}$	Maximum number of search requests by the clients at time t .
$\mathbf{s(t)}$	Maximum number of store requests by the clients at time t .
$\mathbf{\lambda(t)}$	The maximum number of searches/store requests per second at time t , i.e., $\lambda(t) = q(t) + s(t)$
$\mathbf{1/Q_i(t)}$	Maximum search time by the i^{th} node at time t .
$\mathbf{1/S_i(t)}$	Maximum store time by the i^{th} node at time t . Thus the minimum rate of store and search is at time t is $\frac{1}{\frac{1}{Q_i(t)} + \frac{1}{S_i(t)}} = \frac{Q_i(t)S_i(t)}{Q_i(t) + S_i(t)}$ Constraint: This minimum rate of stores and searches must be the better than the requested rate of stores and searches, i.e., $\frac{Q_i(t)S_i(t)}{Q_i(t) + S_i(t)} \geq \lambda$

2.5.2 Memory and its Constraints

The focus of parallel computer development is moving from the processor perspective to the memory system perspective. While the performance of processors in general doubles every 18 months (Moore's law)[50], the performance of memory increases by 8 % each year. Hence a fast processor can spend a lot of time waiting for data from memory. In general, fast memories are small and large memories are slow. This is due to selection delays, and dense storage cells. Hence, very large data sets may be accessed more quickly by dividing them into smaller memories connected to different processors. This leads to the distributed memory parallel computers. In these systems, each processor has its own local memory to which it has direct access; together they form a node. The other processors can only access the data in the local memory of another node through a communication network, by asking the local processor to send or receive the data. Most application programs work with multiple address spaces for each local memory, and for the explicit organization of data transfer between address spaces. This makes parallel computers more difficult to program than their single processor counterparts. Various message passing libraries such as MPI (Message Passing Interface) or PVM (Parallel Virtual Machine) are used to program such systems. Also, key information about the memory requirements of the running tasks, the nature of their accesses to memory, and the frequency with which the memory is searched,

Table 2.2: Memory and its Constraints

Symbol	Description
RAM_i	The total memory available locally at the i^{th} node (Gbytes).
$DB_{i_res}(t)$	The memory (Gbytes) of the i^{th} node reserved to store new profiles and to restore the database for a failed node at time t . $DB_{i_res}(t)$ is a <i>decreasing</i> function as more profiles are added to the database.
$RAM_{overhead}$	The memory overhead (Gbytes) for the operating system and applications. Constraint: The reserved memory and the memory used for database must be less than the total available memory, i.e., $DB_i(t) + DB_{i_res}(t) \leq RAM_i - RAM_{overhead}$

data is inserted or deleted has to be known to effectively model and program the system. Table 2.2 shows variables and constraints defined to characterize the memory requirements of the search system.

2.5.3 The Network and its Constraints

Parallel application performance is the key objective for users of a system. The features and limitations of the underlying messaging and network system have different implications in different applications. Characterization of the system interconnect provides qualitative analysis for comparing various platforms in order to choose the most suitable platform for a given application. Most performance measurements of the communication systems are given in terms of two parameters: Latency and Bandwidth. Latency is the delay or the time for a packet of data to

get from one designated point to another. Bandwidth on the other hand refers to the data rate supported by a network. Table 2.3 defines the parameters that characterizes the communication of messages between nodes.

2.5.4 Goal of DRA

The goal here is to formulate the resource allocation problem using the quantities defined for the parallel search system. The initial distribution problem is:

Given the average values of q_{avg} , s_{avg} of $q(t)$ and $s(t)$ respectively, the time period t_{life} , the capabilities of the nodes (CPU_i , RAM_i), the initial database size $DB(0)$, the network bandwidth L_i and the required search/store rate performance ($M\lambda$), determine the number n of nodes required and the initial distribution of the database $DB(0)$ to the i^{th} node (i.e., $DB_i(0)$) and the RAM reserved $DB_{i_res}(0)$ required on each node to meet database growth requirement and node failures.

That is, given CPU_i , $DB(0)$, RAM_i , $RAM_{overhead}$, q_{avg} , s_{avg} , and L_i determine n , $DB_i(0)$, and $DB_{i_res}(0)$, such that the constraints:

$$\frac{Q_i(t)S_i(t)}{Q_i(t) + S_i(t)} \geq \lambda(t)$$

$$DB_i(t) + DB_{i_res}(t) \leq RAM_i - RAM_{overhead}$$

$$T_{DBm} + T_{build} \leq \Delta T_{\max}$$

Table 2.3: Network and its Constraints

Symbol	Description
L_i	The available effective bandwidth (meaning the rate at which profile data can be transferred) between the i^{th} node and server (Gbps; 1 bps = 1 bit/sec).
$\Delta T_{\max}$	The specified maximum amount of time allowed for a recovery from a node failure.
$T_{DB_m(t)}$	The actual time required to restore the m^{th} node's database on the remaining working nodes. If the m^{th} node fails, its database $DB_m(t)$ is restored from the network drive. That is, a portion $DB_{mi}(t)$ of $DB_m(t)$ is sent to the i^{th} working node in such a way that $DB_m(t) = \sum_{i=1, i \neq m}^n DB_{mi}(t)$ If all nodes have identical resources, then $DB_{mi}(t) = DB_m(t)/(n-1)$ is sent to each of the remaining $n-1$ nodes. The time to transfer the data $DB_{mi}(t)$ from the network drive is: $T_{DB_m} = \sum_{j=1, j \neq m}^n \frac{1}{L_j} \times DB_{mi}(t)$
T_{build}	The total time required to extend the decision tree on each of the nodes to accommodate other additional data $DB_{mi}(t)$. This time is assumed to be proportional to the size of the transferred database, i.e., $T_{\text{build}} = \beta \times \max_i \{DB_{mi}\}$, where β is deduced empirically and $DB_{mi}(t)$ is the portion of the $DB_m(t)$ allocated to the i^{th} node. This assumption has been found to be reasonably valid in practice. Hence the constraint: $T_{DB_m(t)} + T_{\text{build}} \leq \Delta T_{\max}$

are satisfied. The main performance objective is to keep the rate of service of search requests $Q_i(t)$ approximately equal over all nodes 1 through n (represented by i and j), or minimize

$$\sum_{i,j=1, i \neq j}^n (Q_i(t) - Q_j(t))^2.$$

2.6 STATMOND and DRA

Run-time monitoring of the parallel machine is essential for the successful operation of applications and their tasks. The tasks include calls to I/O, access shared data from memory, insertion of data, and computations. Any of these tasks may show a high dynamic behavior and consume more resource time than necessary. Unless run-time monitoring is used to determine the appropriate parameters that affect the system DRA of resources to applications and tasks, it is unlikely that high performance will be attained. STATMOND selectively monitors resource utilization and environmental parameters at regular intervals of time and this information can be dynamically applied to DRA to keep the $Q_i(t)$ approximately the same and respond to failures. This can thereby optimize resource allocation to cluster applications and may achieve smooth progress and performance of all tasks that run on the cluster.

2.7 Summary

This chapter covered in brief the role of the parallel machine at LIT to store profiles and conduct searches for matching profiles in large DNA databases. The use of CODIS database by the FBI, the law enforcement agencies at state and local levels was described in this chapter. A novel method employed by LIT to set up the CODIS database on the parallel machine to perform searches and stores of DNA profiles has been explained. This chapter introduced the problem of DRA on the parallel machine and how the information provided by STATMOND can be used to solve a part of this complex problem.

The next chapter reviews and discusses several existing tools as reported in the literature that achieve objectives similar to or different from STATMOND and that are used to solve DRA problems.

Chapter 3

Survey of Related Tools

3.1 Introduction

There are several tools that provide clients with information about availability of resources in a cluster for both high performance and high throughput computing. A good queuing system and an efficient workload distribution method are desired for a system that processes a lot of independent tasks. But a parallel application with highly communicating tasks needs a method to balance not only CPU load but also network load. There is still room for development of good freely available monitoring systems with low intrusiveness usable in computer clusters; none of the existing were judged to adequately meet the needs of dynamic resource allocation methods in this project.

3.2 Monitoring Tools Developed in Academia

In recent years, computational grid software infrastructures such as Globus[48], Legion[38], Condor[60] and Netsolve[46] have been developed at various research centers. These software infrastructures combine distributed computing and data resources in high performance applications. Most computational grid architectures and clusters employ scheduling features to select the most appropriate resources based on parameters such as CPU load and memory availability. Schedulers decide when and which resources are to be used by various tasks run by the cluster. These scheduling decisions must be based on the performance of each resource that performs the task it has been assigned. The performance of each resource can be measured by systems that are designed and implemented to take periodic measurements about the resource's activity. Numerical models can then be developed to generate forecasts of dynamic performance levels. Several systems that monitor, measure, and forecast dynamic resource performance conditions are reviewed below.

The NWS (short for Network Weather Service) [55, 56, 57] is a tool that provides dynamic resource performance forecasts in grid computing environments (Figure 3.1). Forecasts of CPU availability and network performance are based on measurements taken on a short-term (10 seconds to 60 seconds) basis. These forecasting methods are based upon:

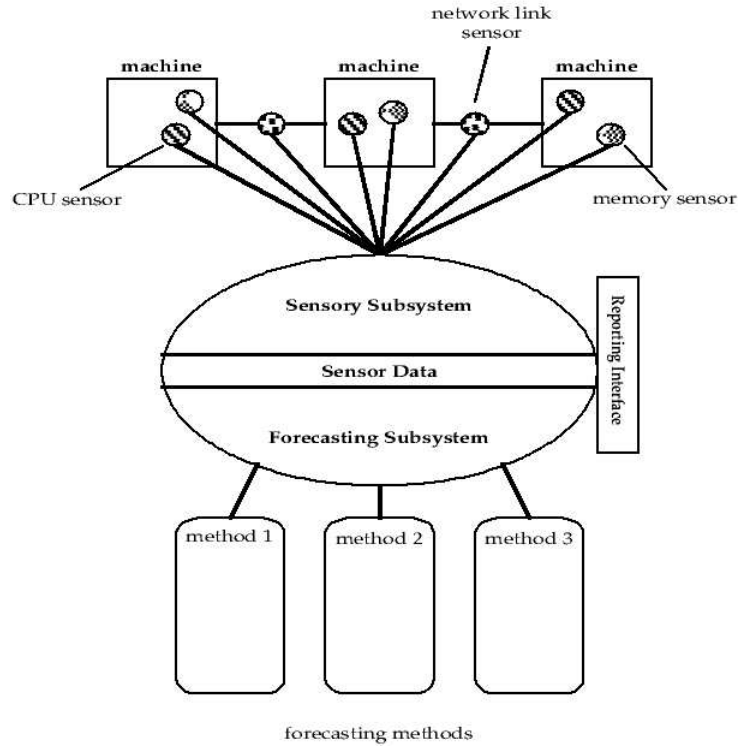


Figure 3.1: The Structure of Network Weather Service[57]

1. mean-based methods on the measurement sample
2. median-based methods
3. autoregressive methods

NWS sensors execute on every host that is monitored. Figure 3.2 shows the monitored hosts A, B, and C that are within the cluster being monitored. NWS is a centralized system, and the measurements are forwarded to a administrative client utility that controls the system. This utility can run on any machine inside

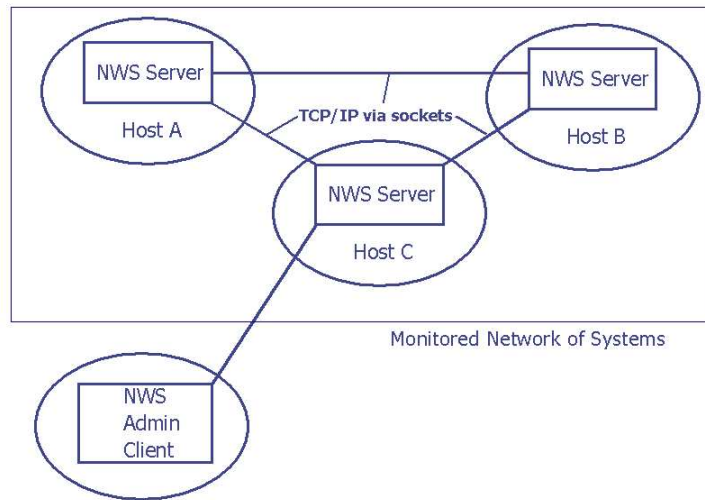


Figure 3.2: NWS Servers Running on Three Monitored Hosts[57]

or outside the monitored network.

NWS operates an arbitrary set of performance sensors and generates forecasts from the periodic readings it obtains. This tool does not use the wide range of information that is available in the `/proc` directory, but is based on conventional Unix utilities such as `vmstat`, which can be a significant source of measurement error. This error is propagated when NWS predicts the availability of the CPU. Any monitoring tool in parallel computing is primarily used by scheduling methods to distribute tasks or jobs. Network Weather Service does not monitor, measure or forecast memory availability on a node, which is vital for resource allocation or load balancing. However, the NWS was enhanced to provide a broader metric set, forecast trend visualization, and asynchronous notification[54]. A new SNMP

sensor has been integrated with NWS that is used to collect data tracked by SNMP agents.

NetLogger (short for Networked Application Logger) [44, 42, 43, 41] attempts to determine the source of performance problems by performing a detailed instrumentation of all software components including the applications, OS, and the nodes. This tool comprises several other tools that generate precision event logs to provide detailed system monitoring, a Java based agent to manage logging of data, and tools to visualize the data on an interface. NetLogger monitors the behavior of the entire communication path between the source and the destination application and produces time-stamped logs of the events at all the critical points of the complex system. Code is inserted in application software that makes calls to the NetLogger application programmer interface (API) to produce event logs. Neither NetLogger nor NWS maintains aggregate information on each node about all the other nodes in the network, which is a key feature of STATMOND. This allows each node to have equal responsibility in implementing the scheduling method.

ClusterProbe [61] is a Java-based tool for monitoring large clusters of workstations. The tool provides a multiple-protocol communication interface that can be connected to various types of external accesses from the clients. ClusterProbe has three main components:

1. The monitoring server
2. The monitoring proxy
3. The agent

The monitoring server handles the requests from various nodes on the network and forwards the results to clients that requested the information. This communication takes place through a communication adaptor. ClusterProbe provides several communication adaptors that support various communication protocols. The monitoring proxy manages a subset of nodes that perform similar tasks. Such a subset can also be constructed for nodes with similar hardware or software configuration. The monitoring proxy provides scalability of the tool. Agents execute as daemons on all nodes that need to be monitored. Each agent collects and reports local resources parameters to the monitoring proxy using a RMI (Remote Method Interface) protocol.

As a Java based monitoring tool, ClusterProbe is an open, flexible, and extensible tool. However, RMI has several disadvantages: (1) It does not provide a full feature set. (2) There is no mechanism for object description. (3) There is no language independence. The resources monitored by ClusterProbe and STATMOND are similar except for a few parameters such as IP and ICMP packets received and sent. While in the STATMOND framework, each node acts as an individual

information source, in ClusterProbe, information is aggregated through the monitoring server. In addition, no detail is mentioned on whether ClusterProbe scales to a large number of computers, without consuming significant system resources.

GARDMON [52], utilizes client-server methodology and provides transparent access between the monitoring server and the nodes on the network. The Gardmon-server is the system utilization provider, whereas the Gardmon-client is responsible for interacting with Gardmon-server and users for data gathering in real-time and presenting information graphically for visualization. The client is implemented with object-oriented, client-server, and Java computing technologies.

The Gardmon architecture can significantly load the server when the number of nodes is large. Also Gardmon does not supply data in a standard format, such as XML, that is accessible by parallel applications or subsystems, except for the gardmon-client.

SCMS [51] is an advanced open source integrated cluster management tool for Beowulf Clusters. SCMS allows users to perform administrative tasks, including software installation and replication of control files. The tool provides a real-time monitoring subsystem, many command tools, and a powerful graphical user interface. Another feature of this tool is its alarm system service that notifies a user with email or by executing a script when a certain condition occurs. This tool uses Java for an effective GUI development framework. However, this tool is

intrusive since it uses a rich set of GUI tools including the Swing Class Library.

3.3 Monitoring Tools Developed in the Industry

Big Brother [4] is a web-based performance monitor that reports availability of software services at each node of a cluster. The most attractive feature of Big Brother is its display, which is available to any web browser. Big Brother supports plugins and extensions, and STATMOND can be integrated with Big Brother so that statistics gathered by STATMOND can be reported to users via Big Brother.

The other features of Big Brother are:

1. Redundancy: Multiple instances of Big Brother can run in parallel; hence a problem is reported even if one monitoring system fails.
2. Platform Independent: Big Brother can run on Unix/Linux and also on NT/Win2K with a scaled down version.
3. Flexibility: The web display is customizable, and alarms and warning bells can be easily redefined.

3.4 Other Monitoring Tools

Presented below are several monitoring tools that obtain a specific measurement.

- Network Related Measurements
 1. Active Measurement Program[1]: Round-trip time, packet loss, and routing.
 2. Surveyor[34]: One-way delay and routing between sites.
 3. Iperf[13]: Maximum end-to-end TCP and UDP bandwidth.
- Application Related Measurements
 1. Prophecy[28]: Average and standard deviation of code segments.
 2. Pablo[21]: Log of data about performance events.
 3. Kappa-PI[16]: Performance analyzer for parallel MPI and PVM programs.
 4. Paradyn[23]: Dynamic instrumentation and performance consultant.
 5. Peridot[26]: Automatic performance analysis tools for teraflop computers.
 6. S-Check[31]: Automated program sensitivity analysis tool.
 7. KOJAK[15]: Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks.

8. DELPHI[6]: An Integrated, Language-Directed Performance Prediction, Measurement, and Analysis Environment.

- Input/Output Activity

1. Pablo[21]: Log of I/O events data.

- CPU

1. Autopilot Performance Monitor[1]: CPU load measurements.
2. ATLAS[3]: CPU usage
3. GRIDVIEW[12]: Grid status information on the web.
4. SvPablo[35]: Hardware performance counters.
5. PAPI[22]: Hardware performance counter.
6. PCL[25]: Hardware performance counter.
7. RABBIT[30]: Hardware performance counter.

- Disks

1. Autopilot[1]: Disk Measurements

- Memory Access

1. MUTT[18]: Tracks memory utilization.

- Tracing Formats

1. JUMPSHOT[14]: CLOG format.
2. UPSHOT/NUPSHOT[2]: ALOG format, Visualization tools for MPI generated trace files.
3. SDDF[32]: Self Defining Data Format for trace files.
4. SIMPLE[33]: Generic event trace analysis environment.
5. PICL[27]: Timestamped trace data on interprocessor communication, processor busy/idle times and simple user-defined events.
6. MPICL[17]: MPI based PICL.
7. ParaGraph[24]: Tool for graphical view the trace data by PICL and MPICL.
8. VAMPIR[36]: An event trace browser.
9. DIMEMAS[7]: Performance prediction by using a VAMPIR trace file of a real program execution.

- Timer

1. PTR[29]: API for measuring intervals of program execution, in terms of wall clock, user CPU, and system CPU time.
2. CARNIVAL[5]: Waiting time analysis.

- Online Process Information Access and Control

1. Falcon[10]: On-line monitoring and steering tools for parallel/distributed applications at the thread-level, based on a custom threads package.
2. OMIS[20]: Portable interface to event-based monitoring.
3. DPCL[8]: On-line monitoring and steering tool.
4. DYNINST[9]: On-line monitoring and steering tool.

3.5 Summary

Clusters and grids are used extensively for research, scientific and commercial applications. As the load of the cluster increases, effective tools to monitor and manage the cluster become necessary for the efficient administration and utilization of the system. Several tools are available in research labs and industries that monitor several parameters such as round-trip time, delays, CPU load, program performance, hardware performance and memory utilization. However, it is not easy to build a tool that meets all requirements for DRA. Moreover, not many tools are available that are flexible and scalable on a network. Though STATMOND has the limitation that it can monitor only Linux based systems, it is decentralized, and it collects kernel and process information directly rather than through intermediate tools provided by the OS. Work still needs to be done

to improve the scalability of STATMOND and this will be discussed in the last chapter.

The next chapter details the STATMOND design and framework as a relatively simple and flexible monitoring system.

Chapter 4

STATMOND Design Overview

4.1 Introduction

This chapter describes STATMOND's mechanisms that probe, forward and gather information, and provide the basic building blocks to implement an effective monitoring system. The high level design of the essential features is discussed. Specifically, those features that enable STATMOND to be simple, flexible, and scalable are highlighted.

4.2 STATMOND Process

STATMOND, as the name suggests, is a service-daemon, which runs in the background on all the nodes of the cluster that are to be monitored. The basic ar-

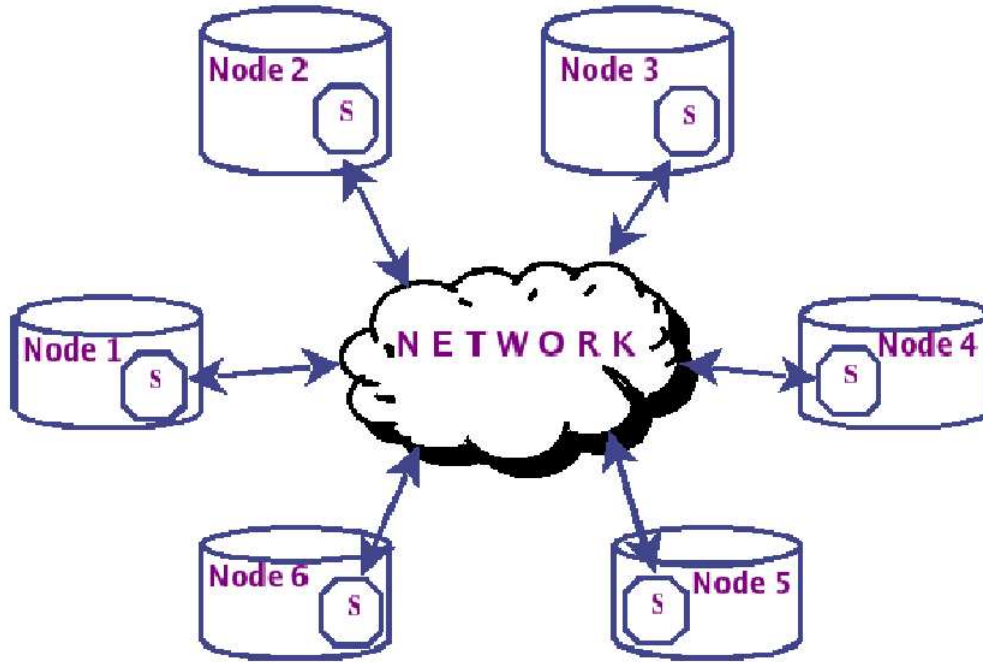


Figure 4.1: STATMOND Framework with 6 Nodes

chitecture is shown in Figure 4.1. The nodes can be independent workstations or a cluster of servers within the same subnet connected through a low-latency, high-bandwidth interconnect. The system shares measurement data among all participating nodes (those running STATMOND), enabling application or system software on any node to access a local copy of all measurement data.

STATMOND on each node has 5 main functions.

1. STATMOND periodically probes the node for kernel and process information and stores the data in a local data structure. The time interval between 2 probes is user-defined and can be anywhere from a few seconds to a few

minutes. However, the time interval is chosen as large as it is reasonable to expect changes in the information that is collected to minimize STATMOND resource utilization.

2. STATMOND forwards collected information to other nodes as it becomes available. The data structure is accessed, and information is sent to designated nodes thorough the communications medium.
3. STATMOND receives and interprets the information from other nodes, and updates the local data structure. The information received is categorized based on the node and the time at which it was sent. The data structure is locked while the update takes place.
4. STATMOND receives and responds to special information requests clients such as system administration and application software. Specific information may be requested from time-to-time on an as-needed basis. STATMOND processes these requests while it carries out its other functions.
5. STATMOND logs the entire information in the local data structure to a file of user-selected format. This file exists on the local machine and can be accessed by several tasks, applications or methods that need the data.

4.3 STATMOND Kernel Instrumentation

STATMOND collects data from the kernel's data structures made available through the `/proc` virtual file system of Linux. The `/proc` file system is located off the root directory in `/proc` and is the user's interface to the system's kernel. The `/proc` file system is not a standard file system; it provides a hierarchically structured method for access to information provided by device drivers, network resources, and the kernel. Since the `/proc` file system does not occupy space on the hard drive, it is referred to as a "virtual file system"; its "files" are actually interfaces to reporting software in the kernel and device drivers. In fact, the "files" use access points to kernel and device driver structures that are maintained in the memory.

The `/proc` directory has several read-only files and sub-directories that provide both system and process information. A few system specific files and subdirectories that STATMOND uses to collect data are:

1. `/proc/cpuinfo`: Provides information about the collection of CPU and system architecture-dependent items with a different list for each supported architecture.
2. `Loadavg`: 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, 15 minutes.

3. Meminfo: Reports the amount of free and used memory in bytes (both physical and swap) on the system as well as the shared memory and buffers used by the kernel.
4. Stat: Reports the number of "jiffies" (1/100ths of a second) that the system spent in user mode, user mode with low priority (nice), system mode, and idle.
5. Uptime: Reports the uptime of the system (seconds), and the amount of time spent in the idle process (seconds).
6. /proc/devices: Shows the available devices.
7. /proc/interrupts: Provides directions for interrupt usage.
8. /proc/iomem: Provides a memory map of system's memory and various devices.
9. /proc/ioports: Provides input/output (I/O) port usage and status.
10. /proc/kmsg: Shows the kernel messages
11. /proc/meminfo: Provides memory related information.
12. /proc/modules: Provides information about loaded kernel software modules.
13. /proc/net: Provides statistics about packets received and transmitted.

14. `/proc/version`: Provides kernel and system version data.

The other category of sub-directories helps users fetch process-specific information. Each running process has a sub-directory under `/proc`, named after its process id (PID). A few files in this category are:

1. `/proc/PID/cmdline`: Provides the command line with its arguments.
2. `/proc/PID/cwd`: Provides the current working directory of the running process.
3. `/proc/PID/envIRON`: Provides values of environment variables available to the process.
4. `/proc/PID/mem`: Provides information about memory allocated to the process.
5. `/proc/PID/status`: Provides the status of the process, such as sleep mode and run mode.
6. `/proc/PID/statm`: Provides process memory utilization data.

For example, to get the status of a running process, the file `/proc/PID/status` can be read. Most Linux utilities collect data from the `/proc` directory. The 'ps' command, for instance, uses the `/proc` file system to obtain information about

the running processes on the system. STATMOND does not gather information from standard Linux utilities such as `vmstat` and `top` for two reasons. First, the executable size of Linux utilities is large since they provide more than the required functionality. Second, some information from the utilities is not as easily accessible as it is from the `/proc` directory. By using information in the `/proc` filesystem, STATMOND eliminates the "middleman" and goes directly to the source. The advantage of this approach is efficiency; the disadvantage, however, is that the method is specific to Linux.

4.4 STATMOND Data

The current implementation of STATMOND supports the instrumentation of the following information:

1. Kernel data: Aggregate and per-CPU data, including:
 - (a) Measures of percentages of time spent in user, low-priority user, system, and idle modes,
 - (b) Measures of network activity on a per-NIC (network interface card) basis.
 - (c) Measures of real and virtual memory utilization.
 - (d) Measures of page faults (read and written).

- (e) Measures of swap space utilization (read and written).
 - (f) Measures of input and output to data storage devices.
2. System configuration data: Number of CPUs and their performance, the size of available disk storage, and the configuration of network connections.
 3. Process data: This is mainly the process execution information such as the PID, %CPU used by the process, and the memory occupied by a process. STATMOND collects the data from the /proc/PID subdirectory of the /proc file system. This output along with the PID, is used to identify one or more specific executing processes to be monitored. Once a process on a host is identified, a request can be made to execute a script to that host (typically written to execute under a shell, Perl, or a similar utility). Such a script must produce one or more data items separated by white space on its standard output, and these data items are appended to the reports provided by the reporting STATMOND process. Because of the scripting capability, an authentication procedure is utilized to ensure that only digitally signed STATMOND processes may request execution of a script and collection of new data fields.

The STATMOND process formats measurement data as a table (Figure 4.2). One row is added whenever the process receives data from a host for the first

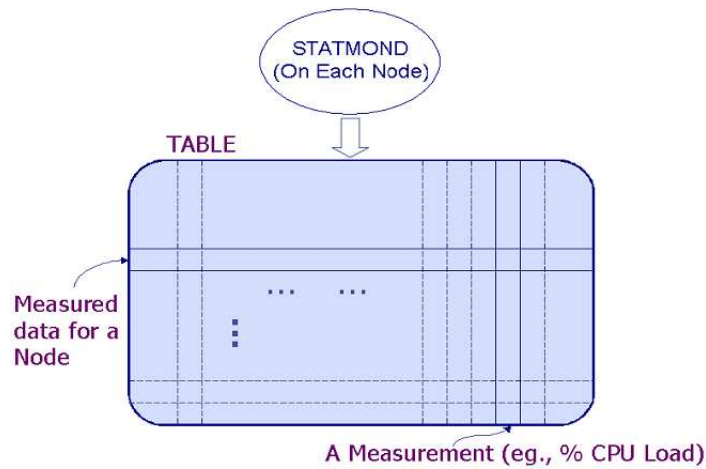


Figure 4.2: General STATMOND Table Structure

time. Each row is time-stamped with the date and time at which it received the last update from a host. Each column is labelled by a unique identifier for one type of measurement, such as CPU1TEMP for the temperature of CPU 1. The STATMOND process periodically sends its most recent measurements for its local host to other nodes, and it can be configured to save its table data to a file in either HTML or XML format, and/or to respond to interprocess queries on the local host. The saved file is updated by the STATMOND process on a periodic basis, and the period can be specified on its command line.

4.5 Data Transfer Between STATMOND Processes

STATMOND employs multicast network communications. Multicast allows a single message transmitted from one node to be received by several hosts dispersed over the network. The network hardware, rather than the transmitting host, replicates the transmission so that all hosts receive it. Communications take place through a socket. A socket is a network endpoint associated with a process that communicates with another endpoint (socket) associated with another process. Both sockets can lie on the same machine, or they can be on different nodes. Communication between any two processes through their respective sockets takes place in an agreed-upon format called a protocol.

Traffic should be sent as efficiently as possible - the less bandwidth used, the better. Without multicast technology, unicast (point-to-point) communication is available, but this is, we suspect not sufficient for the current problem. TELNET, FTP, SMTP and HTTP are unicast-based protocols with one source and one destination. To send to multiple destinations, different communication paths are needed between the source and each of the destinations (see Figure 4.3). Therefore, a copy of each update is made by the transmitting node and sent separately to each receiver. This does not scale well to a network with a large number of

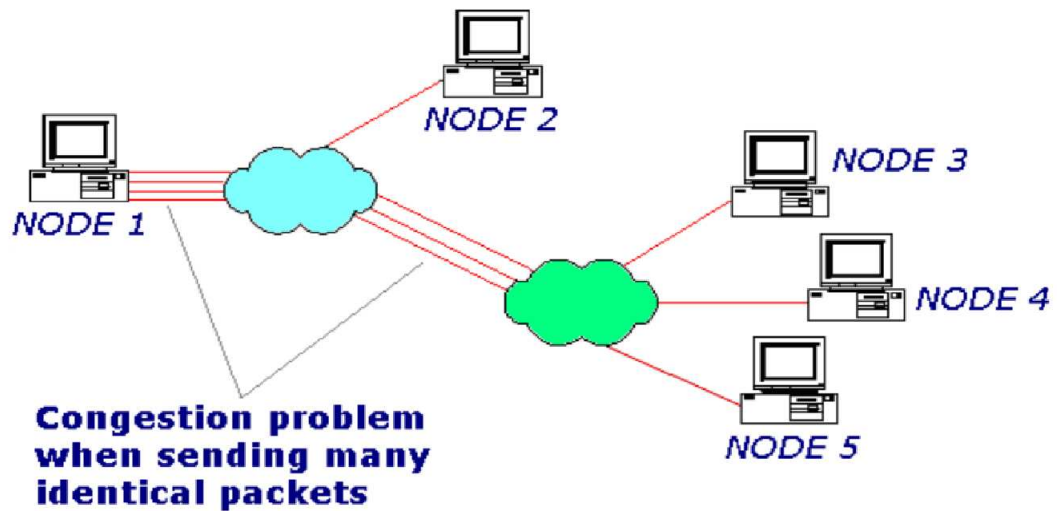


Figure 4.3: Unicast Loads the Sender

hosts. TCP (short for Transmission Control Protocol) cannot be used in multicast applications, as it is unicast-oriented. UDP (short for User Datagram Protocol), on the other hand, does not require a sender-receiver channel and offers a direct way to send and receive datagrams over a network.

An IP address is a 32 bit integer that uniquely identifies a computer (host) on a network. The IP address is usually divided into four one-byte unsigned integer values (from 0 through 255, inclusive), and these are written as decimal integers separated by periods ('.'). A sample IP address is: 128.69.54.87.

The range of IP addresses is divided into classes, based on the first eight bits (leftmost decimal integer of its dotted notation) of the IP address. Multicast ad-

addresses are class D addresses: these addresses start with the first three bits set to one and the fourth set to zero. This means multicast addresses range from 224.0.0.0 to 239.255.255.255, and multicast traffic can be received by, or transmitted from, hosts in this range of addresses. Each address, analogous to a radio frequency, identifies a different and specific multicast group to which listening nodes can "tune" as shown in Figure 4.4. Some of these multicast addresses are well-known and reserved for a specific purpose. For instance, 224.0.0.1 is the all-hosts group. Any multicast-capable host must join this group at start-up on all its multicast-capable interfaces. 224.0.0.2 is the all-routers group for network routers, and so on. Multicast applications never send datagrams to addresses 224.0.0.0 through 224.0.0.255, as they won't be forwarded across multicast routers. Similarly, groups 239.0.0.0 through 239.255.255.255 should be avoided, as they are reserved to administratively limit multicast traffic to a defined topological region.

As shown in Figure 4.5, when a group is multicast enabled a single packet is sent which is replicated by the group router and forwarded to all the nodes in the network.

4.6 The STATMOND-Client

In the STATMOND monitoring framework, the client (Figure 4.6) refers to any system within the subnet that wants to obtain information from STATMOND

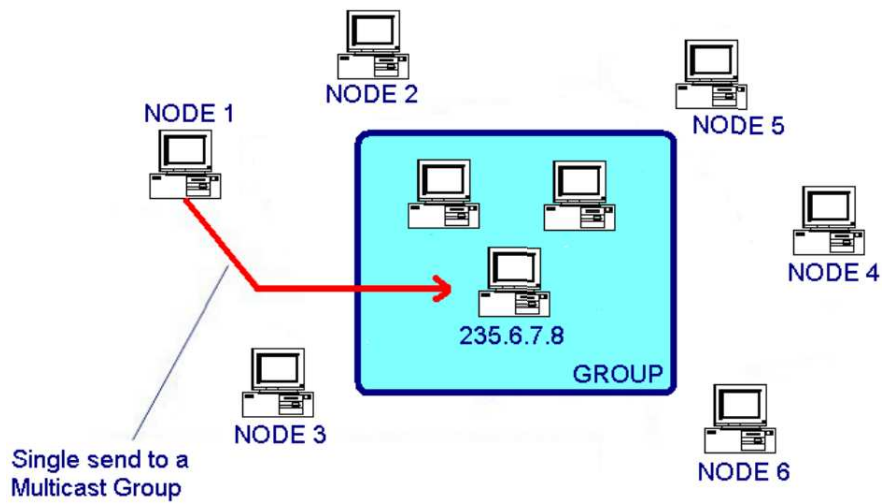


Figure 4.4: Sending Packet to a Multicast Group

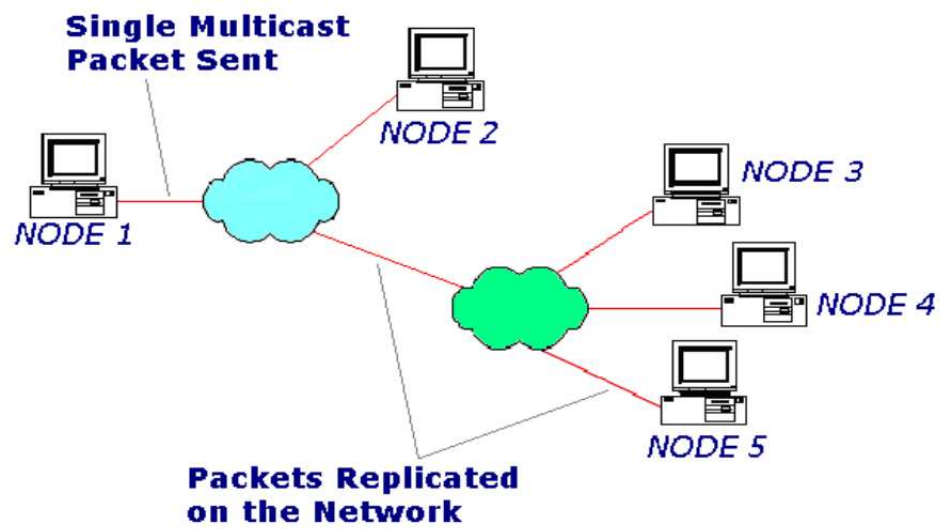


Figure 4.5: Multicast Enables the Sender to Send Just Once

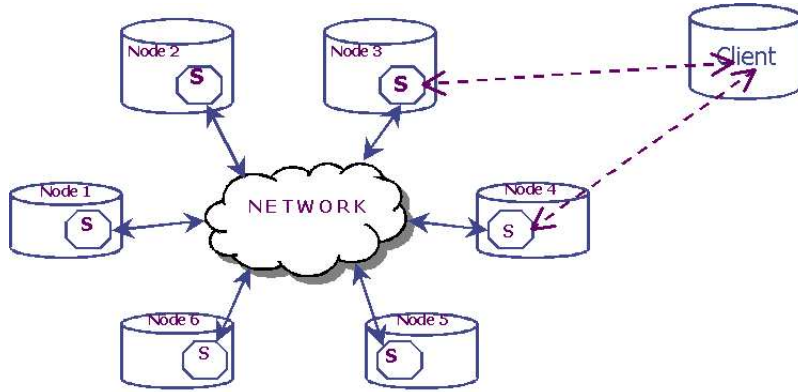


Figure 4.6: Client-STATMOND Interaction

about the status and performance of the nodes on the cluster. The client can also be loaded on any of the nodes in the cluster. The node on which it is loaded may or may not be a functional part of the cluster, and therefore the client is completely independent of STATMOND and its architecture.

However, the client has to follow certain rules to talk to STATMOND. The client establishes an independent connection with STATMOND. This lets STATMOND carry out its other functions in parallel without interruption. However, the information requested by the client is given top priority. For such communications the TCP/IP suite of protocols is used. TCP is designed to perform well for connections that are held for a long time, and transfer a lot of data. TCP offers reliable, connection oriented transmission for upper layer protocols by establishing a connection between source and destination machines and terminating the

connection when transmission is complete. The protocol offers error checking of packets, acknowledgement of successful receipt of packets after error checking and packet sequencing. TCP performs two additional functions to prevent slowdown of the network:

1. Flow control: ensuring that a fast sender (client) does not bombard the relatively slower receiver (STATMOND) with packets.
2. Congestion control: slowing or increasing transmission speed to match total load (or congestion) of the Internet paths separating the client and STATMOND.

4.7 Summary

The design of STATMOND as a distributed architecture is presented in this chapter. Any node on the network can be monitored from other nodes that belong to a group. A detailed description of the kernel and process information that is collected from the `/proc` filesystem of the Linux operating system is also laid out. This process is observed to be faster than the use of Linux tools such as `vmstat` or `top` to acquire data. A special section is dedicated to the UDP suite of sockets along with the multicast protocol for data transfer. Multicast ensures that only one packet of information is sent irrespective of the number of nodes

on the network. The data structure that is maintained as a table in the memory is introduced and will be discussed in detail in the next chapter. This data structure is important as it is the only source of aggregate data in the memory. Moreover, it has to be simultaneously accessed by all functions of STATMOND. This data structure is flexible, and parameters can be added or deleted without major changes in the code. This makes STATMOND quite extensible in terms of the parameters it monitors.

This chapter also brings out the importance of independent clients that talk to STATMOND for specific information. As such, the clients use TCP/IP to connect with STATMOND for a reliable connection. The next chapter describes how the various features of STATMOND are implemented in code and highlights any problems that were encountered during the implementation.

Chapter 5

Implementation of STATMOND

5.1 Introduction

This chapter presents a detailed design of the STATMOND server and client. It performs an analysis of the monitoring framework through a thread-level abstraction. Threads are lightweight processes and sequences of a program that perform certain functions within the program. Threads allow programs to break down their processing into smaller “chunks.” This chapter also discusses the data handling and forwarding mechanism through the use of data structures and custom filters in STATMOND. Hence a clear picture of the implementation details and various design features incorporated into the system is obtained.

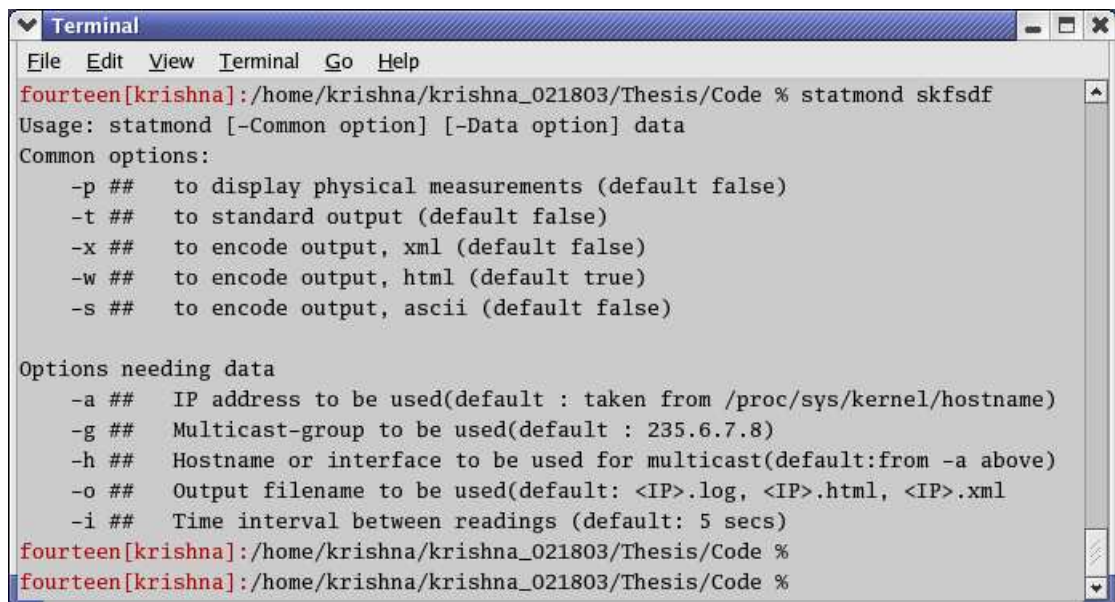
5.2 Detailed Design of STATMOND

Unix processes can execute in either the foreground or background. A process that runs in the foreground interacts with the user at a terminal or console through I/O communication, whereas a background process runs without user involvement. The user does not directly perceive its presence (or a display), but can check its status and interact with it using an agreed-upon protocol. As mentioned before, the term ‘daemon’ is used for processes that perform service in background. STATMOND begins execution when loaded (run at command line) by the user or other system software. It runs forever and does not die or get restarted. It operates in the background, waits for requests to arrive, processes these requests and possibly responds by returning information to the software that generated the requests. It can spawn other processes to handle multiple client requests. STATMOND’s command set includes the several options listed below (also see Figures 5.1 and 5.2). When executed correctly STATMOND runs in the background and one way its PID can be determined is shown in Figure 5.3.

Usage: statmond [-Common option] [-Data option] data

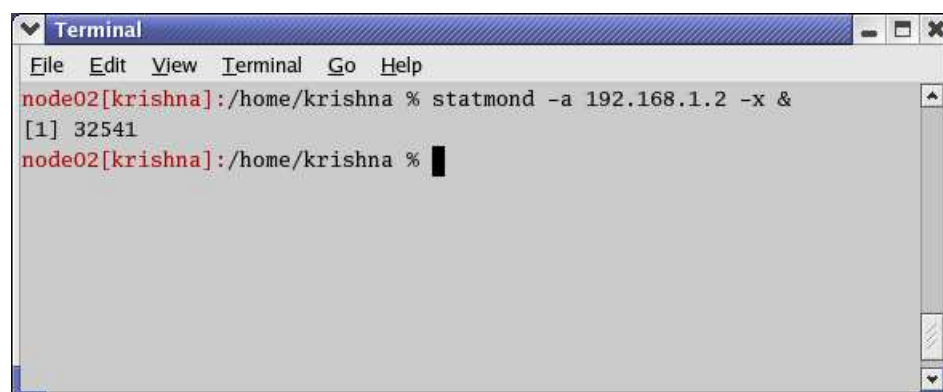
Common options:

- p : to display physical measurements (default false)
- t : to standard output (default false)



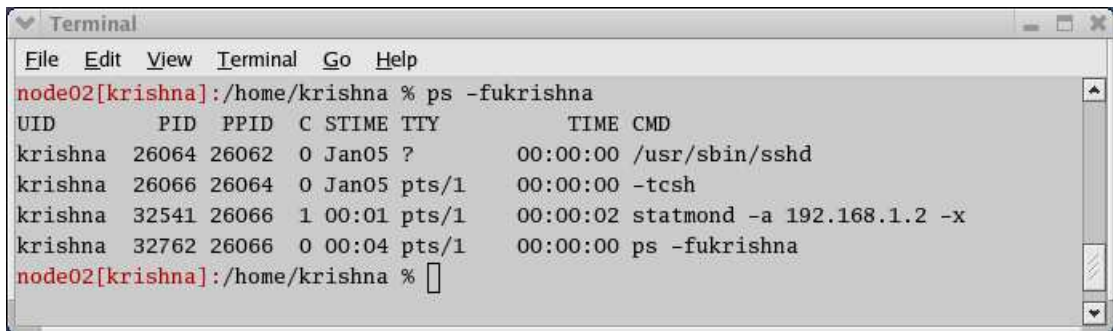
```
Terminal
File Edit View Terminal Go Help
fourteen[krishna]:/home/krishna/krishna_021803/Thesis/Code % statmond skfsdf
Usage: statmond [-Common option] [-Data option] data
Common options:
  -p ## to display physical measurements (default false)
  -t ## to standard output (default false)
  -x ## to encode output, xml (default false)
  -w ## to encode output, html (default true)
  -s ## to encode output, ascii (default false)
Options needing data
  -a ## IP address to be used(default : taken from /proc/sys/kernel/hostname)
  -g ## Multicast-group to be used(default : 235.6.7.8)
  -h ## Hostname or interface to be used for multicast(default:from -a above)
  -o ## Output filename to be used(default: <IP>.log, <IP>.html, <IP>.xml)
  -i ## Time interval between readings (default: 5 secs)
fourteen[krishna]:/home/krishna/krishna_021803/Thesis/Code %
fourteen[krishna]:/home/krishna/krishna_021803/Thesis/Code %
```

Figure 5.1: Incorrect Command Line Execution of STATMOND



```
Terminal
File Edit View Terminal Go Help
node02[krishna]:/home/krishna % statmond -a 192.168.1.2 -x &
[1] 32541
node02[krishna]:/home/krishna %
```

Figure 5.2: A Correct Command Line Execution of STATMOND



```
node02[krishna]:/home/krishna % ps -fukrishna
UID      PID  PPID  C  STIME TTY      TIME CMD
krishna  26064 26062  0  Jan05 ?        00:00:00 /usr/sbin/sshd
krishna  26066 26064  0  Jan05 pts/1    00:00:00 -tcsh
krishna  32541 26066  1  00:01 pts/1    00:00:02 statmond -a 192.168.1.2 -x
krishna  32762 26066  0  00:04 pts/1    00:00:00 ps -fukrishna
node02[krishna]:/home/krishna %
```

Figure 5.3: STATMOND's PID

-x : to encode output, xml (default false)

-w : to encode output, html (default false)

-s : to encode output, ascii (default false)

Data options:

-a : IP address to be used(default : taken from /proc/sys/kernel/hostname)

-g : Multicast-group to be used(default : 235.6.7.8)

-h : Hostname or interface to be used for multicast(default:from -a above)

-o : Output filename to be used with suitable suffix

(default: <IP>.log, <IP>.html, <IP>.xml)

-i : Time interval between successive probes (default: 5 secs)

STATMOND's design is symmetric: All hosts have the same statistics on the population of hosts from which they are able to receive data, although typically such information is delayed by different amounts of time at each host, and all hosts execute the same STATMOND software. The hosts participate in a multicast group defined by the monitoring daemon processes. Each host periodically sends information to the multicast group in order to distribute its local information to the group's members, regardless of the group's size. The network switches replicate, distribute, and eliminate the need for each host to send point-to-point message traffic to every other host. Each host must, however, read message traffic from all the other hosts, and this is a limitation to the current implementation of STATMOND. A possible solution to this limitation is proposed and under investigation, which combines multicast and unicast UDP message protocols to adaptively route aggregate data throughout a cluster.

STATMOND, has to determine the IP address it will use for its communications to and from other nodes when it starts. This is because a node may be connected to more than one LANs, each through an interface. An IP address is assigned on a per-interface basis, so a node may possess several IP addresses, as many as the interfaces. For example, a node with both Fast Ethernet and Gigabit Ethernet will have at least one IP address for each interface. The node is therefore known by multiple addresses, and every service on this host can be referred to by

multiple names. STATMOND picks up the IP address that was used to join the multicast group as the reference address. However, the exact ramifications of this effect depend heavily on the network design. In particular, careless design can result in a node becoming reachable by one address but not by another.

The activities of STATMOND as a daemon can be conceptually divided into 2 parts that will be handled in the next couple of sections:

1. Data structure that stores values for the monitored parameters
2. The methods that act on the data structure to carry out STATMOND's functions.

5.2.1 Data Handling in STATMOND

STATMOND uses the concepts of object-oriented programming to implement its architecture. The STATMOND data structure is an object of the Parameters class that contains and manages its own internal data (parameters) and provides well-defined interface services that allow STATMOND functions to use the parameters. One characteristic of the object-oriented style is that each parameter is contained within and managed by a Parameters object declared from the Parameters Class. This object is an association of the various parameters with the operations that manipulate that them. The Parameters object's internal state

affects how it delivers services to other such objects or functions but, other functions and objects cannot directly manipulate these internal data structures. The internal data of the Parameters object can only be altered indirectly by invoking one of the services associated with that the Parameters object.

Each STATMOND process on the network has only one Parameters object with its own internal data, but all the instances of a Parameters class on the network provide the same set of services. The Parameters class implements a serialize function that converts the entire data contained in the object into a stream of bytes. STATMOND, in order to notify other nodes of its current state, forwards this stream through the network(s) using a multicast protocol. A receiving STATMOND unserializes the stream into a new Parameters object. This method enables the statistics of nodes to be transferred on a network as contiguous bits of data. Each such received stream is identified by the time stamp and the IP address of the node that sent it. STATMOND generates a map in the memory for the Parameters objects of remote nodes received across the network with their corresponding IP addresses. Each node has a map that consists of values for each of the parameters objects. This map is sorted by the IP addresses of remote parameter objects. Due to delays experienced during the probe and transfer of data, each system may receive a different stream corresponding to one sender node. Thus each node may have a different parameter object for a remote node

and the same instant of time.

Many programming languages (such as SmallTalk, C++, Ada9X) include features that can support the object-oriented style to store, forward and receive data, but C++ is preferred in this implementation. This is because C++ has the ability to interact with the kernel of the Linux OS. Also, C++ provides a powerful library (Standard Template Library(STL)) to insert or delete data, or to allocate/reallocate memory dynamically and perform other operations. Moreover, C++ allows STATMOND to perform more than one task at a time, through a design described in the next section. This enables various tasks of STATMOND to cooperate for system resources instead of competing for them.

5.2.2 Multithreading in STATMOND

Computers in the early days used to run a single-task or a single job at a time. Once a job was completed, the next job was started and so on. Most computer programmers were frustrated by such batch-oriented processing, and computer vendors developed multitasking operating systems. Multitasking refers to a computer's ability to execute multiple jobs concurrently. Most modern operating systems such as Linux and Windows have the ability to run more than one program at the same time. While users download a big file through a web browser, they can play their favorite game; both programs run at the same time. Multithreading

extends the multitasking paradigm. Instead of multiple programs, multithreading involves multiple threads of control within a single program. Hence the operating system runs multiple programs, and each program runs multiple threads of control within the program.

A thread is a single sequence of execution within a program. It is a lightweight process and requires relatively lesser memory and startup time than a usual process. Each process can include many threads; limitations are unique to each OS. All threads of a process share memory that includes program code and global data. In fact, they share the entire working environment such as the current directory, socket descriptors, and user ID. However, each thread is identified by a unique Thread ID, and threads communicate with each other through the shared memory.

The STATMOND program involves four distinct threads as shown in Figure 5.4.

The Probe Thread (Figure 5.5) runs in the background to collect the measurements. The thread populates the Parameters object if the values are different from the previous measurement. The thread inserts the updated object to a map that maintains the statistics of all nodes identified by the IP address. The probe thread invokes the send thread to notify the nodes about its current status. The thread then goes to sleep until it needs to probe again. This probe time interval

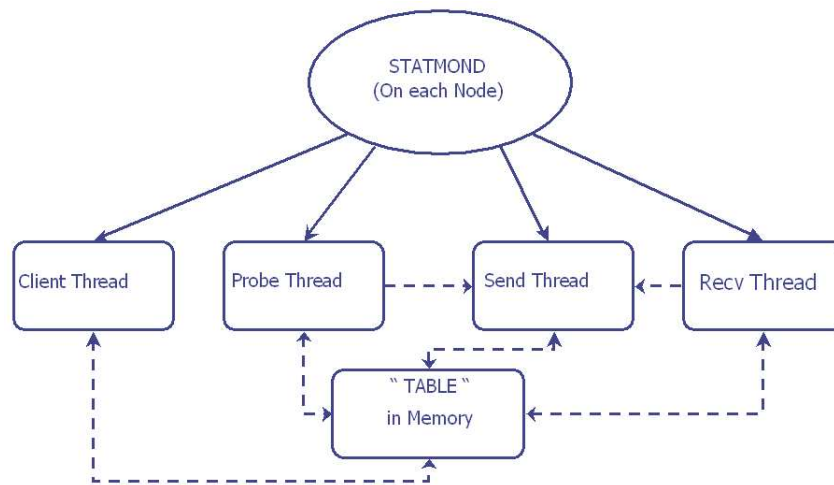


Figure 5.4: Internal Software Structure of STATMOND

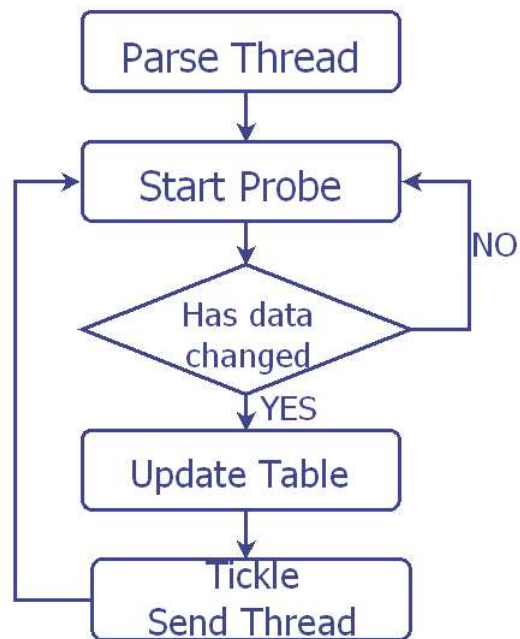


Figure 5.5: The Probe Thread

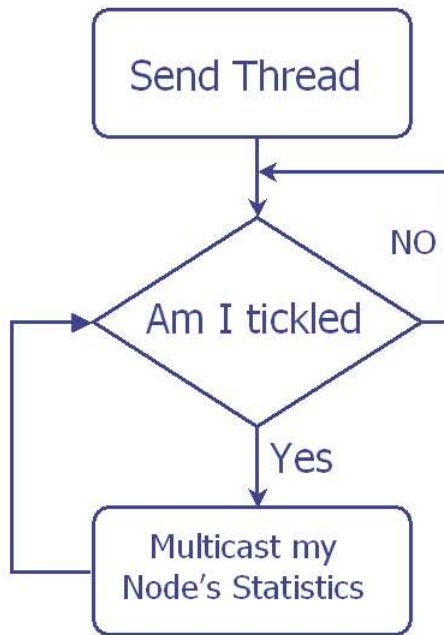


Figure 5.6: The Send Thread

can be set by the user on the command line.

The Send Thread (Figure 5.6), sends the serialized measurements as a byte stream on a pre-determined interface. Since the send thread uses UDP sockets it does not wait for an acknowledgement from the receiver. It goes into the sleep mode and waits there to receive the next call from the probe thread. It is a costly operation in terms of time and memory, to create a new thread whenever data needs to be sent onto the network. The send thread hence prefers to wait in sleep mode until the probe thread call it.

The Receive Thread (Figure 5.7) receives the serialized data and stores the IP

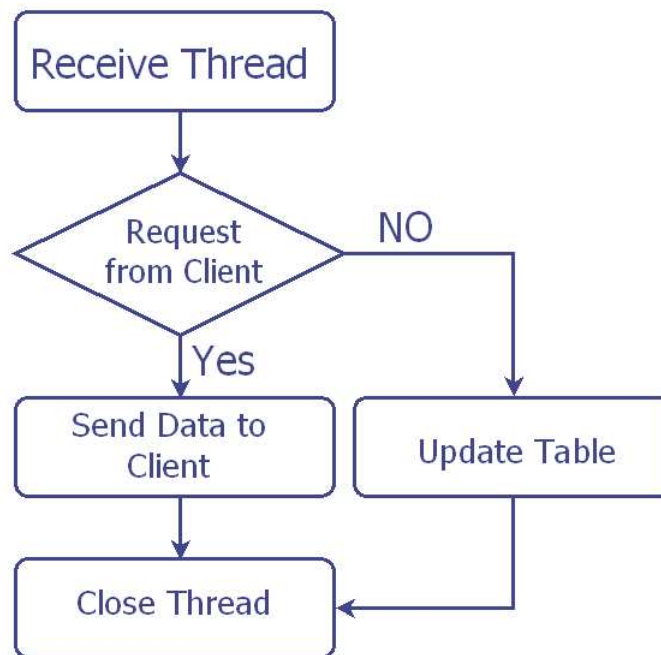


Figure 5.7: The Send Thread

address from which the byte stream was sent. It also assigns a time stamp at the time of arrival. This time stamp is compared with the time stamp at which the sender forwards the packet to determine the delay in the network and also to find out if the data is reasonably current. This packet is unserialized and the contents are stored in the map at a position referenced by the sender's IP address.

The Client Thread (Figure 5.8) responds to specific requests regarding a particular parameter made by a user present on any node of the subnet. The data is sent in the order requested. However, the client thread is memoryless. It does not wait until it receives all requests from the client. It responds to every single

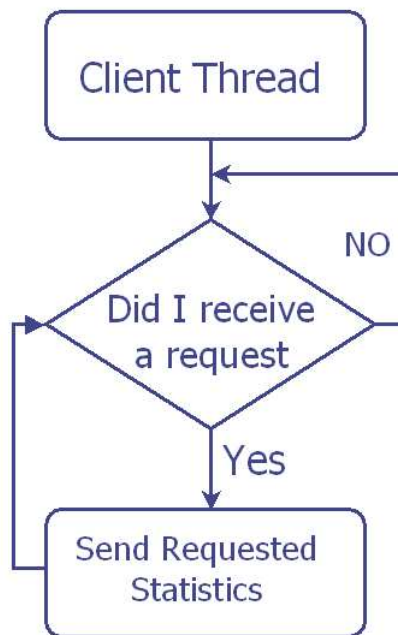


Figure 5.8: The Client Thread

request separately through a new TCP/IP connection.

5.2.3 Multicasting through STATMOND

STATMOND as described before uses multicast to deliver content to multiple receivers (Section 4.5, Page 48) . As such, STATMOND acts as multicast receiver and sender, and so data flow is full duplex. As a receiver STATMOND:

1. Gets a UDP socket,
2. Binds to the application's port number,

3. Joins the multicast address group,
4. Receives the packets, and
5. Waits for more packets.

Alternately, STATMOND as a sender:

1. Gets a UDP socket, and
2. Sets the IP Time-To-Live (TTL) as desired by user at command line. The TTL has a double significance in multicast. First, it controls the live time of the datagram to avoid being routed forever due to routing errors. Second, TTL limits how far multicast traffic will expand across routers. Every router has TTL threshold assigned to each of its interfaces. Only datagrams greater than the interface's threshold are forwarded. For example, a datagram with $TTL < 128$ may be restricted to the same continent while a $TTL = 1$ is restricted to the same subnet.
3. Sends the packet to the multicast address and port number, and
4. Waits for more packets to be sent when complete.

Though multicast is a mature technology, there are areas that need improvement:

1. There is no established standard for reliable data delivery via multicast. Several implementations exist since different applications have different multicast requirements.
2. There is no standard way to confirm whether the local router is multicast enabled. Even though the router responds to an ICMP Router Discovery Message (RFC 1256), this is not a guarantee that the router is configured to support multicast.
3. There is no standard way to determine neighbors in a group. Hence, two groups on a subnet can have a conflict if they use the same multicast address.
4. There exist no diagnostic or debugging tools that determine problems in case multicast fails. This can cause users to never find out what's wrong with the setup.

STATMOND still has several hurdles before it is a fully scalable tool. In spite of the bottlenecks, multicast is convenient for any application or service that requires several nodes to interact. It is difficult to scale the size of the network without the savings multicast provides.

5.2.4 STATMOND's Output Representation

STATMOND uses the eXtensible Markup Language (XML) to output its data. XML adds structure and conveys information about documents and data. XML supplies information by ordering the document with "elements" comprising a "start tag", some content (optional), and an "end tag". XML is similar in concept to HTML, but whereas HTML conveys graphical information about a document, XML represents structured data in a document.

STATMOND's XML output does not contain information on how the data contained within the content is rendered. An "eXtensible Style Language" (XSL) style sheet achieves the graphical rendering of the XML document. XSL style sheets can not only render XML information graphically but can also transform XML documents into other formats. STATMOND's XML output is designed to be easily interpreted and used by other software applications using standard XML parsing methods.

The XML output document is made up of elements that consist of textual information contained between a "start tag", and an "end tag". The parameters' information between the "start tag" and the "end tag" is sometimes referred to as the "content" (or sometimes the "data"). Unlike HTML, the set of tags is not pre-defined, and all elements must be either at the top level, or be contained within another unique element; that is, the document structure is a directed tree

of elements with a single "root" element.

STATMOND chooses the best set of tags for a particular purpose. When STATMOND responds to a client one or more tags may not be present. Information represented in the form of an XML document can be useful to assist local processing on the node without incurring a round trip transaction with another node. In addition, an XML representation of information is useful for passing data between nodes. Because XML parsers are available for many platforms, and if a suitable XML vocabulary is defined for the information, then the information can be passed between several nodes regardless of the underlying platform or operating environment.

A sample STATMOND output when all the nodes (24) are up is shown in Figure 5.9. Each node represented by its IP address has information concealed in its tree. For example NODE03 (IP Address: 192.168.1.3) holds the information as in Figure 5.10. A much detailed information is obtained by expanding the remaining tags (Figure 5.11). NODE03 has just one processor and data corresponding to this processor is obtained by parsing the /proc directory. However, this is not the case with multi-processor nodes. Information relevant to each and every processor of a multi-processor node is made available by STATMOND to the user (Figure 5.12).

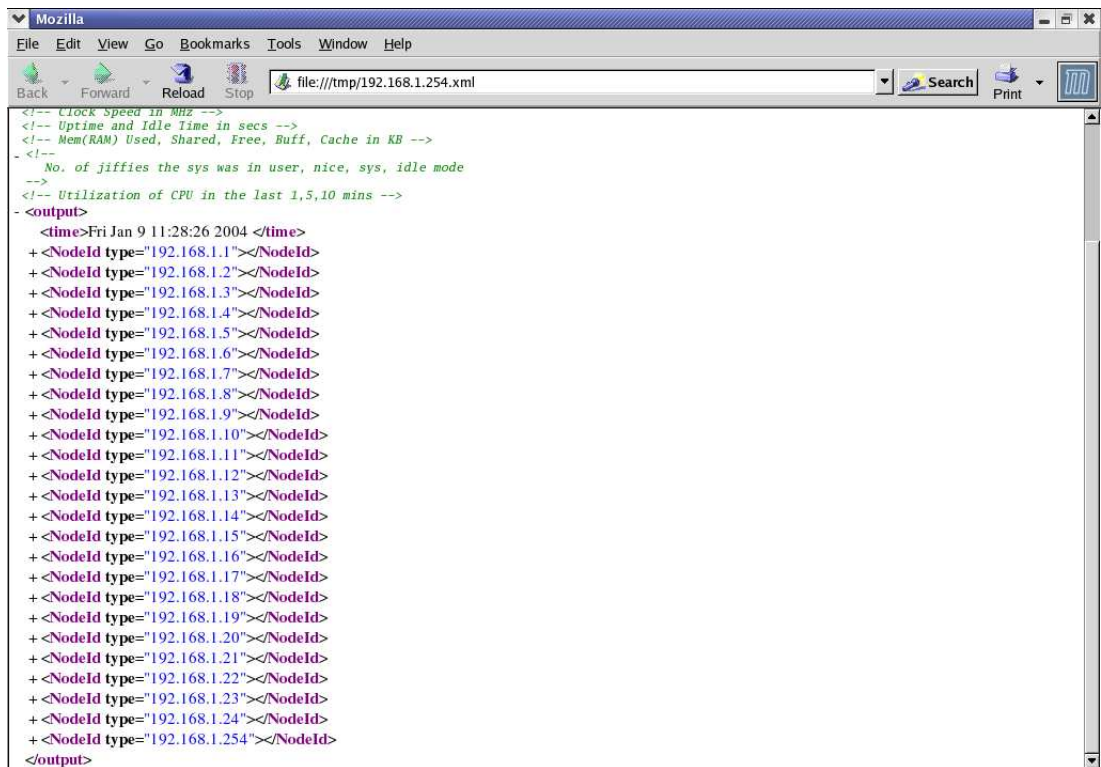
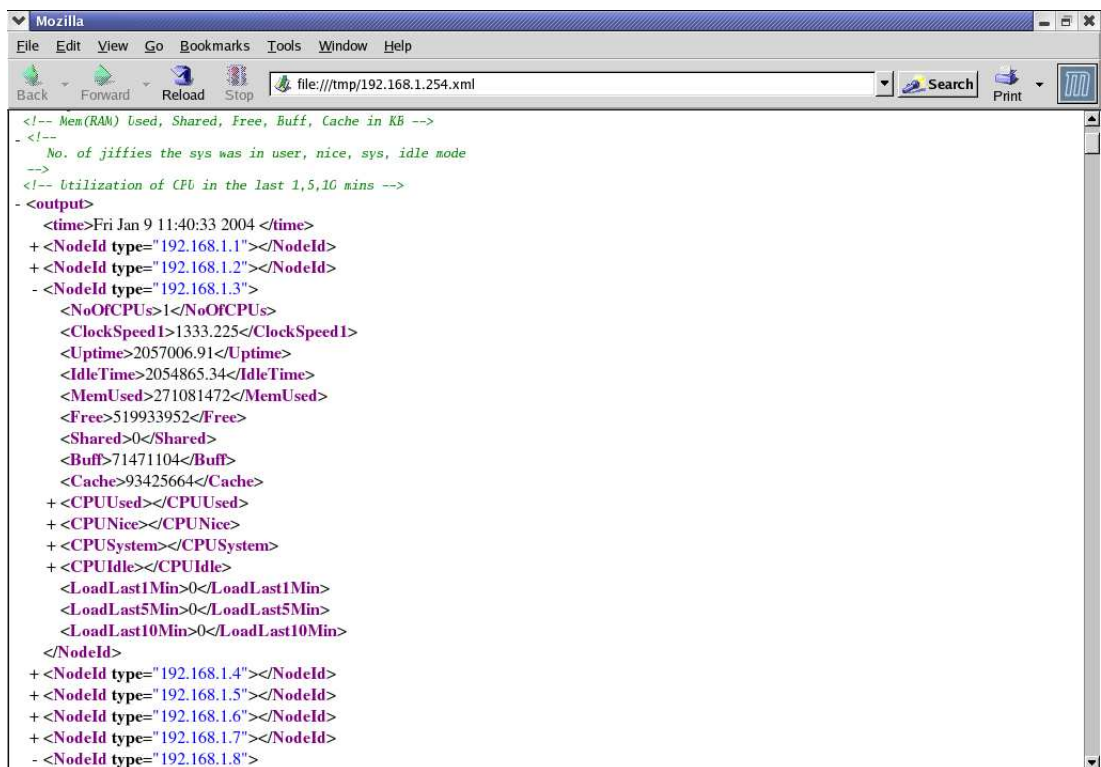


Figure 5.9: Snapshot of XML Data File Contents



```

<!-- Mem(RAM) Used, Shared, Free, Buff, Cache in KB -->
- <!--
  No. of jiffies the sys was in user, nice, sys, idle mode
-->
<!-- Utilization of CPU in the last 1,5,10 mins -->
- <output>
  <time>Fri Jan 9 11:40:33 2004 </time>
+ <NodeId type="192.168.1.1"></NodeId>
+ <NodeId type="192.168.1.2"></NodeId>
- <NodeId type="192.168.1.3">
  <NoOfCPUs>1</NoOfCPUs>
  <ClockSpeed I>1333.225</ClockSpeed I>
  <Uptime>2057006.91</Uptime>
  <IdleTime>2054865.34</IdleTime>
  <MemUsed>271081472</MemUsed>
  <Free>519933952</Free>
  <Shared>0</Shared>
  <Buff>71471104</Buff>
  <Cache>93425664</Cache>
+ <CPUUsed></CPUUsed>
+ <CPUNice></CPUNice>
+ <CPUSystem></CPUSystem>
+ <CPUIdle></CPUIdle>
  <LoadLast1Min>0</LoadLast1Min>
  <LoadLast5Min>0</LoadLast5Min>
  <LoadLast10Min>0</LoadLast10Min>
  </NodeId>
+ <NodeId type="192.168.1.4"></NodeId>
+ <NodeId type="192.168.1.5"></NodeId>
+ <NodeId type="192.168.1.6"></NodeId>
+ <NodeId type="192.168.1.7"></NodeId>
- <NodeId type="192.168.1.8">

```

Figure 5.10: XML Data File Contents of a Uni-processor Node

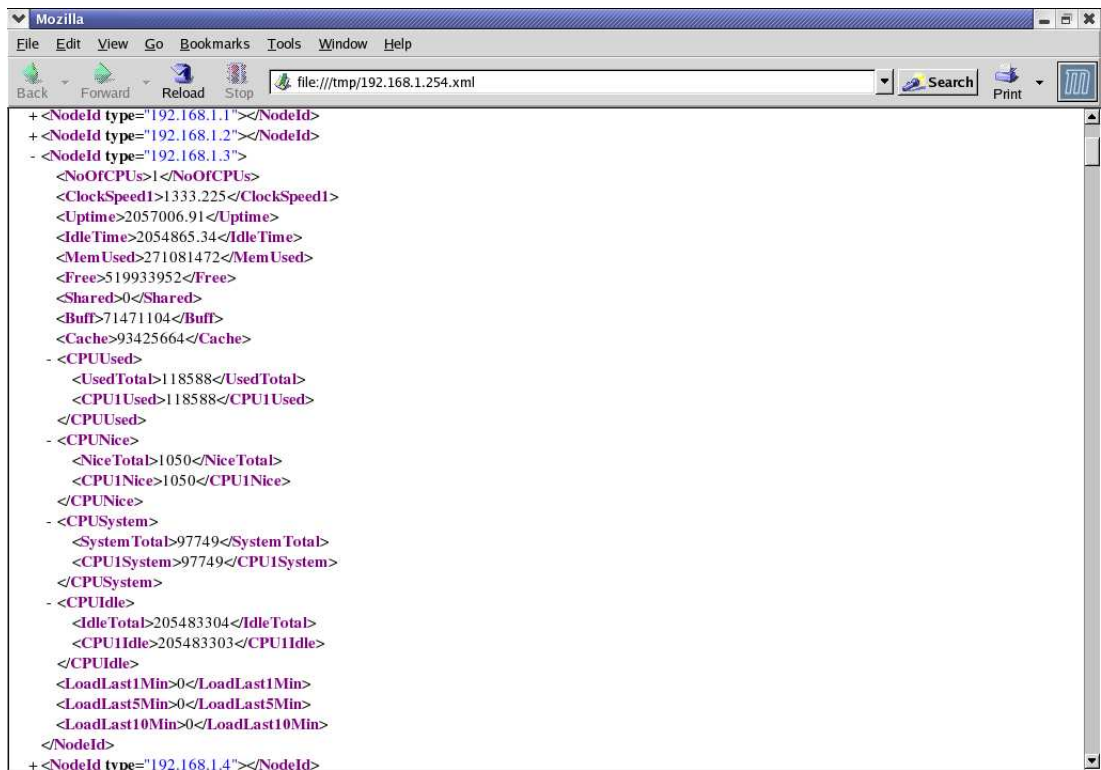


Figure 5.11: XML Data File Detailed Contents of a Uni-processor Node

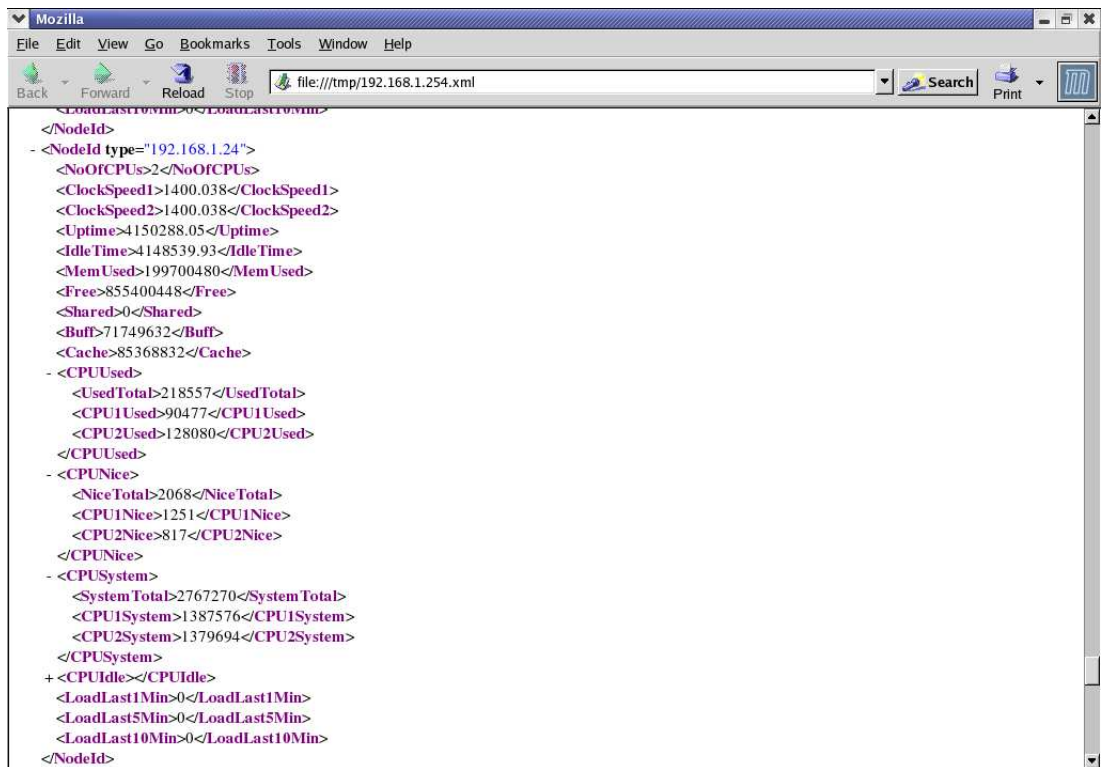


Figure 5.12: XML Data File Detailed Contents of a Multi-processor Node

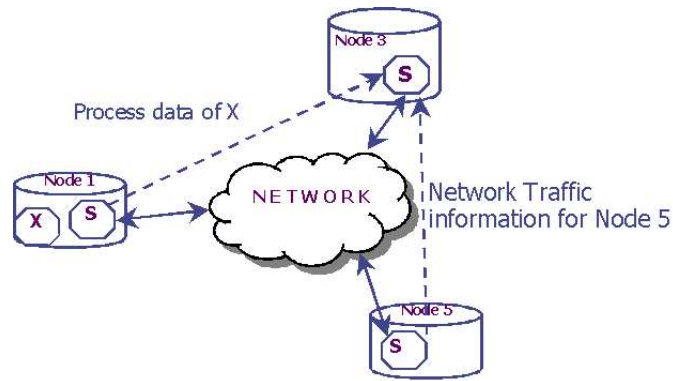


Figure 5.13: Custom Filter - STATMOND Interaction

5.2.5 Custom Filters

In addition to standard measurement functions, STATMOND uses dynamic, user-defined filters for complex, application-specific conditions. These customized filters work invisibly with the system to influence the event-collection process (Figure 5.13). The filters are implemented to minimize this impact on system performances, in which case they can be used on all nodes with minimal system impact. Flexible custom utilities can be used to monitor and generate reports based on any user-defined fields. Reports are in XML format and can be easily modified, viewed, and printed.

Two such sample filters are:

```
getProcInfo <Server> <PortNo> u username e executable
```

getProcInfo is the name of the filter and the corresponding script. This

```

Terminal
File Edit View Terminal Go Help
rpcuser 640 1 0 2003 ? 00:00:00 rpc.statd
root 698 1 0 2003 ? 00:00:00 [rpciod]
root 699 1 0 2003 ? 00:00:00 [lockd]
root 716 1 0 2003 ? 00:00:19 ypbind
root 763 1 0 2003 ? 00:00:01 /usr/sbin/sshd
root 777 1 0 2003 ? 00:00:00 xinetd -stayalive -reuse -pidfil
ntp 791 1 0 2003 ? 00:00:41 ntpd -U ntp
root 803 1 0 2003 ? 00:00:00 /bin/sh /usr/bin/safe_mysqld --d
root 815 1 0 2003 ? 00:00:01 crond
mysql 843 803 0 2003 ? 00:00:10 /usr/libexec/mysqld --defaults-f
xfs 878 1 0 2003 ? 00:00:00 xfs -droppriv -daemon
daemon 896 1 0 2003 ? 00:00:01 /usr/sbin/atd
root 903 1 0 2003 tty1 00:00:00 /sbin/mingetty tty1
root 904 1 0 2003 tty2 00:00:00 /sbin/mingetty tty2
root 905 1 0 2003 tty3 00:00:00 /sbin/mingetty tty3
root 906 1 0 2003 tty4 00:00:00 /sbin/mingetty tty4
root 907 1 0 2003 tty5 00:00:00 /sbin/mingetty tty5
root 908 1 0 2003 tty6 00:00:00 /sbin/mingetty tty6
krishna 10431 1 2 19:16 ? 00:02:57 statmond -a 192.168.1.10 -x
root 22176 763 0 21:30 ? 00:00:00 /usr/sbin/sshd
krishna 22181 22176 0 21:30 ? 00:00:00 /usr/sbin/sshd
krishna 22183 22181 0 21:30 pts/1 00:00:00 -tcsh
krishna 23007 22183 0 21:40 pts/1 00:00:00 ps -ef
node10[krishna]:/home/krishna %

```

Figure 5.14: Processes on Node10 (192.168.1.10)

script specifies the executable and user names of the process (example, Figure 5.14 whose information needs to be obtained periodically. get-ProcInfo communicates with STATMOND on the remote host when executed (example, Figure 5.15). The STATMOND on the remote host extracts the information about the process from the /proc directory and periodically sends the process data to the STATMOND on the user's node. The process data includes allocated memory, user time used, system time used, and total time used for that process. The STATMOND on the user's node dumps this data into an XML file for its use by another application (Figure 5.16).

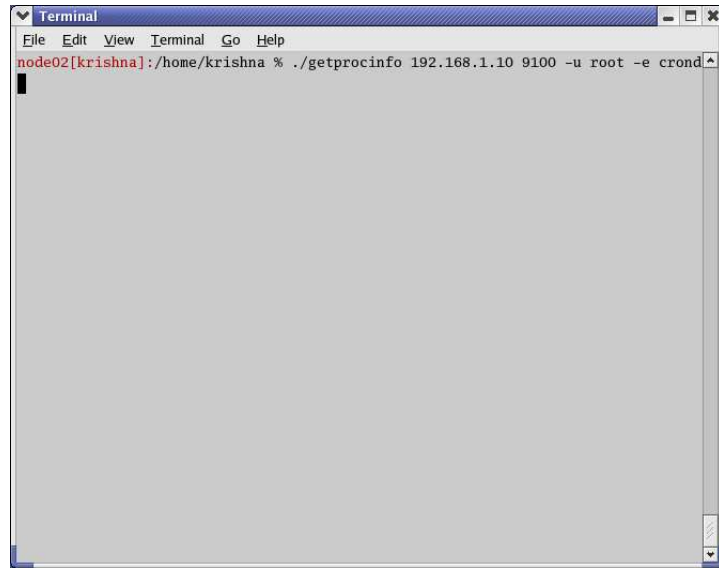


Figure 5.15: Command Line Execution of Custom Filter

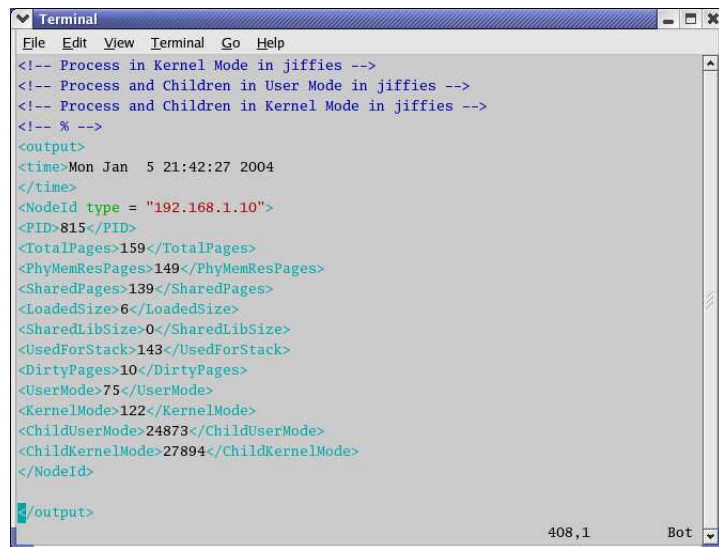


Figure 5.16: XML Data File of Custom Filter

getParamInfo <Server> <PortNo> <Param1> ... <ParamN>

getParamInfo is another filter similar to getProcInfo. However, this filter is used to acquire packet information such as recvpkts, and sendpkts periodically from a remote host. This information is particularly useful to determine the bandwidth utilization of the remote host. These values are also returned and stored in XML format with a time stamp.

5.3 Detailed Design of the Client

STATMOND Client is a program that when executed fetches requested data from a node and stores it in a XML file. This communication takes place over TCP/IP sockets. A user (system administrator or a developer) executes the client program at the command line prompt. The user sets the IP address of the receiver (which acts as a server in this case), and the port on which STATMOND listens for requests at the receiver as arguments for the client program. The user also passes the various parameters for which values are desired. The usage of the client program is as shown below.

Usage: ./client <server> <portno> <param1> <param2> ... <paramN>

The Parameters:

cpus : Find the # of CPUs on the server

clockspeed : Find the clock speed of the server

memused : Find the total RAM used by the server

memfree : Find the memory free on the server

memshared : Find the shared memory

membuff : Find the memory in the buffer

memcached : Find the memory cached

cpuuser : Find the # of jiffies (defined as 1/100 of a second) the server has been in user mode

cpunice : Find the # of jiffies the server is in low priority mode

cpusystem : Find the # of jiffies the server is in system mode

cpuidle : Find the # of jiffies the server is in idle mode

uptime : Find the # of seconds for which the server has been up

idletime : Find the # of seconds for which the server was in idle state

last1min : Utilization of the processor in the last 1 Min

last5min : Utilization of the processor in the last 5 Min

last10min : Utilization of the processor in the last 10 Min

recvpkts : Find the number of packets received

sentpkts : Find the number of packets sent

recvbytes : Find the number of bytes received

sentbytes : Find the number of bytes sent

The server (STATMOND) receives the parameters as a series of strings. The server validates the parameters and collects their values from the data structure. Specifically, the client thread of STATMOND retrieves the values and returns them in the order requested by the client. This organized information is sent as a stream of bytes through the respective TCP/IP connection. The client dumps the received values into an XML file for use by other tasks or applications on that node (Figure 5.17).

A single STATMOND server can receive multiple requests at the same time. More than one client thread may be spawned to process multiple requests from different clients. Both the server and client maintain a list of all connections and the respective values of parameters processed for a particular connection. A system administrator can use this information to monitor a particular activity on a node at any instant.

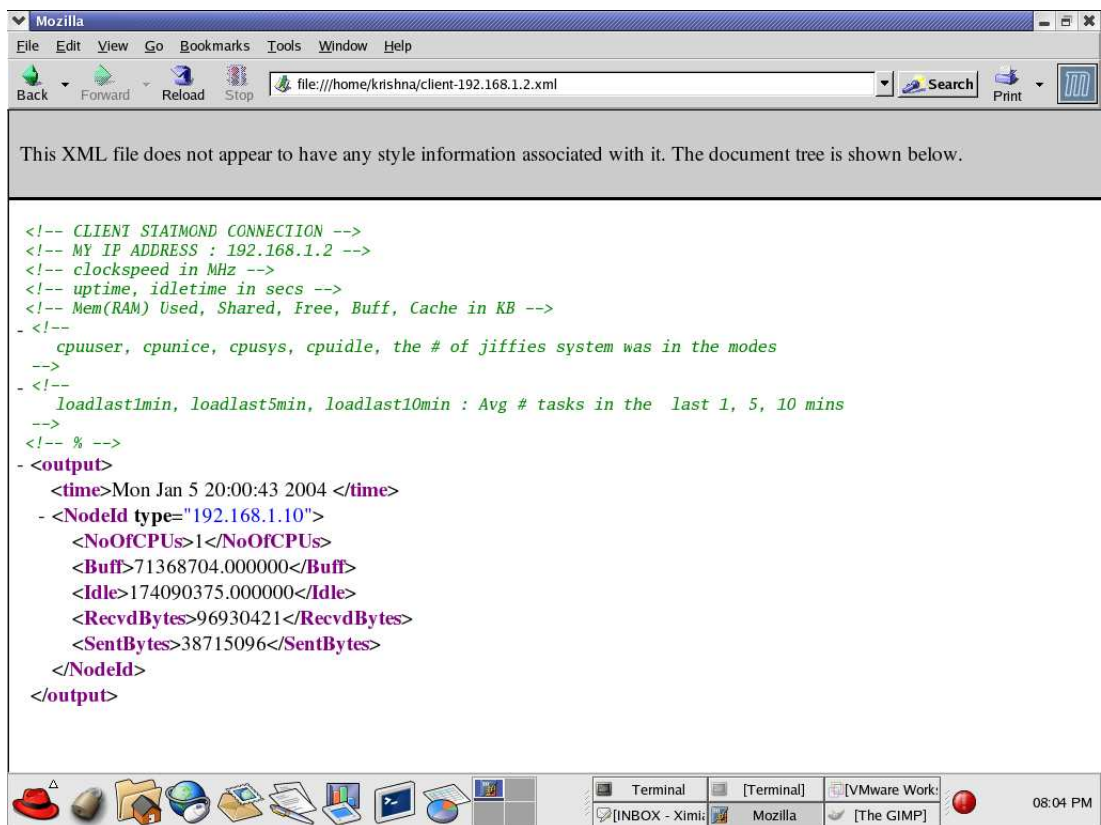


Figure 5.17: XML Data File for Client

5.4 Summary

STAMOND is implemented using a multithreaded architecture. This helps the system to remain fast, promotes extensibility and flexibility, and provides an easy mechanism for reporting results in user-requested formats. Client may connect with STATMOND to obtain more specific information from time to time. The next chapter presents a review of the work performed. An analysis of the system is given and possible future avenues of work to further extend and improve the capabilities of STATMOND are suggested.

Chapter 6

Analysis and Future Work

6.1 Introduction

The previous chapter explains STATMOND's implementation, the detailed instructions for its use, and its sample outputs. This chapter presents an analysis of the tool and assesses whether the goals and requirements of the project have been met. The chapter also suggests the future work that may extend and improve the system.

6.2 Review of Work

Good measurement and management tools are necessary in exploiting computer clusters as platforms for high performance computing. With Linux leading the

pack as the most popular operating system for clusters, new hardware and software tools are being developed to meet the needs and to push the capability limits of clusters. This thesis, however, addresses the lack of status and performance measurement tools, which renders resource allocation for parallel computers more tedious. An accurate status and performance diagnosis of a parallel machine is one of the most important tasks. The goals of this thesis were to design and implement a scalable and reliable system that will measure the key parameters of a Linux based parallel computer in a manner that provides the available measurement data on all nodes and whose operation on running nodes is not sensitive to other nodes' failures.

STATMOND is designed and implemented to serve as a status monitor for a parallel computer. Several versions of this software have been tested. The software uses the IP multicast network protocol, which utilizes features of network switches to replicate transmitted data packets so that a single transmission from one computer can be received by all computers in a multicast group. The software has been tested on the 24 node parallel machine in the Laboratory for Information Technologies. The software is currently using both Fast Ethernet and Gigabit Ethernet LANs for communication. Multicast substantially lowers the CPU overhead to maintain status information on a large population of machines and implements a distributed monitoring system where all hosts have information

on all other hosts.

STATMOND is a peer-to-peer network tool. Each node maintains its own table of the existing conditions for all nodes of the network. The information about the existing conditions is collected from the files located in the `/proc` filesystem. Though the format and structure of the tables across all the nodes is the same, the data that they hold might differ based on the probe time interval, communication errors, and network delays. If the probed data on a node have changed, the current dynamic status of that node is multicast to all other nodes on the network. In this way, the measurement data is ubiquitous and is present at all the potential sites on the network. Moreover, the STATMOND processes running in the background on each node all use exactly the same software and thus provide identical features and functionality. Even if one or more nodes fail, the measurement process is not terminated. The STATMOND processes on the remaining nodes continue to communicate. However, STATMOND tables do not reflect the current data of the failed nodes unless they (failed nodes) rejoin the multicast group. Since the data from each node are time-stamped, failed nodes can be identified by their old data on any other node. STATMOND typically stores the data from its table in a XML file in the `/tmp` directory, and this provides a standard interface between the STATMOND process, and other programs that may run on the parallel computer.

6.3 Analysis of STATMOND

The main goal in developing a new status monitoring tool is to achieve a decentralized, detailed, flexible, and non-intrusive information collection and reporting system, whose services can be used for dynamic resource allocation. The statistics are collected, delivered, and received accurately in a timely fashion with acceptably low levels of system consumption. The current features and functionality of STATMOND are analyzed in the next subsections.

6.3.1 Non-Intrusiveness

The parallel cluster at LIT is built to perform computationally intensive database searches. A 0.5% CPU consumption by STATMOND can affect the time taken by a task perform a search on the same machine. It is essential that STATMOND meet the CPU, communication, and memory monitoring requirements without affecting the regular activities of the cluster.

STATMOND when run on all the 24 nodes of the network consumes less than 0.1% of the total memory available. This conclusion is drawn after several memory usage observations taken over 3 days time. The observations were taken from the `/proc/STATMOND-PID` directory as well as the `top` utility provided by Linux. The observations showed that the peak memory consumption is 0.1 %. This is primarily due to the changes in the space occupied by the data structure of

STATMOND.

The typical message size sent or received by STATMOND is less than or equal to 160 bytes. All parameters (20 in number) are stored in 'double' format (64 bits). Hence, 3840 bytes bytes transmitted to the network by the 24 nodes every probe time interval (for example every 5, 10, or 15 secs). Thus the data rate for a 5 second interval is 768 bytes/sec or 6124 bits/sec. Multicast reduces the use of bandwidth by enabling STATMOND to send an update packet just once to communicate to all other nodes, rather than sending a packet to each node sequentially. This reduces bandwidth utilization by approximately a factor of two. Multicast also reduces excess CPU utilization in sending more than one packet and prevents congestion in the network due to a larger volume of packets.

The CPU utilization of STATMOND is less than 1% when STATMOND does not store the output (information in data structure) in XML, HTML, or ASCII files (Figure 6.1). It is intuitive that STATMOND will consume more CPU cycles to store output in XML format than otherwise. The plots (Figures 6.2 and 6.3) show STATMOND's CPU usage (y axis) on NODE 01 (IP:192.168.1.1) and its dependence on the number of nodes (x axis).

These plots indicate that the CPU resources required to run STATMOND scale linearly with cluster size when output has to be stored. An increase in the number of nodes increases the number of packets received to perform updates to the

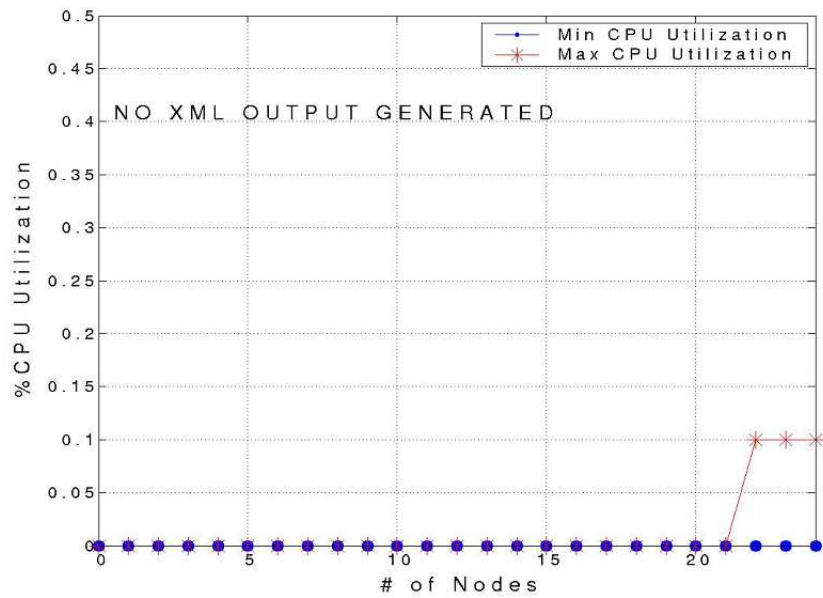


Figure 6.1: %CPU usage on NODE01, No output stored

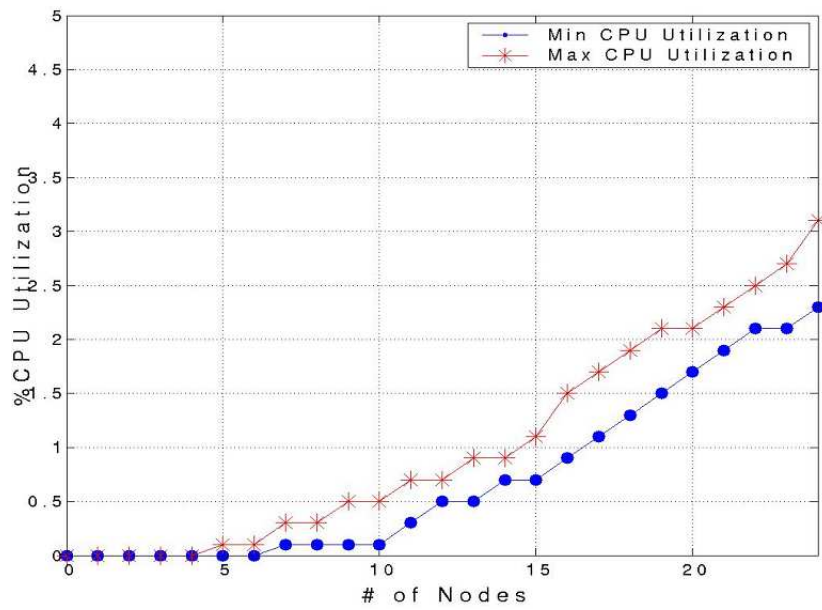


Figure 6.2: %STATMOND NODE 01 CPU usage, Monitoring Interval: 5 secs

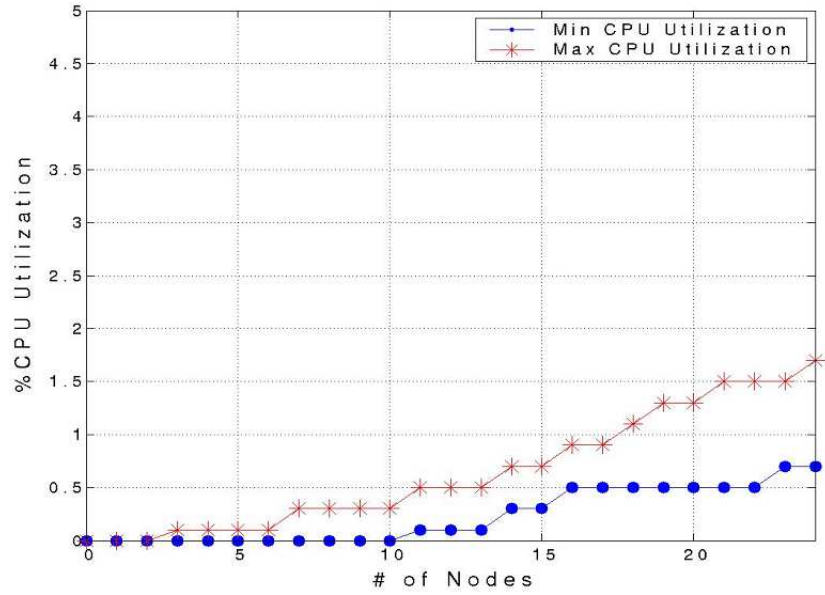


Figure 6.3: %STATMOND NODE 01 CPU usage, Monitoring Interval: 15 secs

data structure and the output file. This causes the extra utilization of the CPU to perform updates and stores to the XML file. For a 5 second update interval, STATMOND requires approximately 0.1% of a CPU on each node per monitored CPU. An objective of future research work is to develop efficient storage of data during regular intervals of time. Another objective is to develop an adaptive messaging strategy that allows CPU utilization to scale logarithmically with cluster size. This will be briefly discussed in the next section. Figures 6.4 and 6.5 show CPU usage on a dual processor machine NODE00 (IP:192.168.1.254). The CPU usage does not improve significantly; this suggests that an efficient method to store output in XML form needs to be developed.

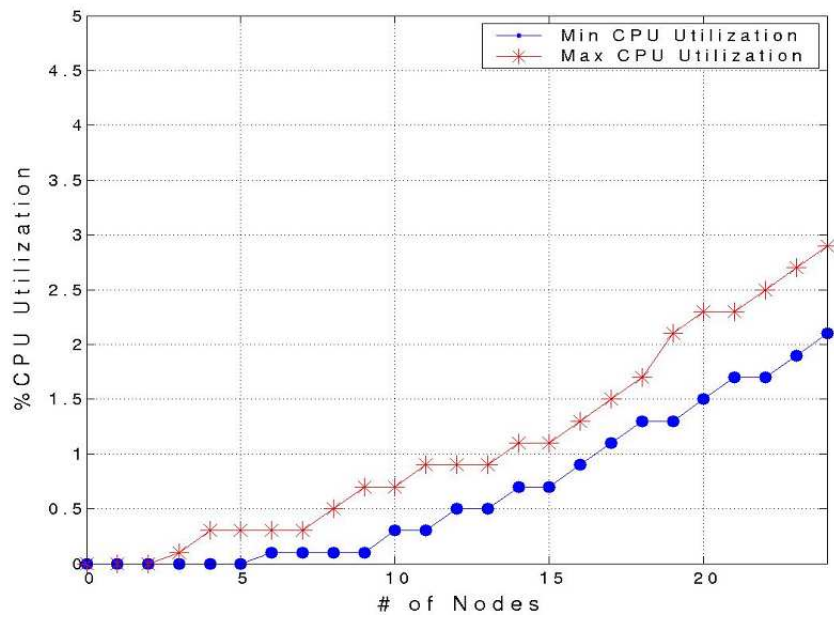


Figure 6.4: %STATMOND NODE 00 CPU usage, Monitoring Interval: 5 secs

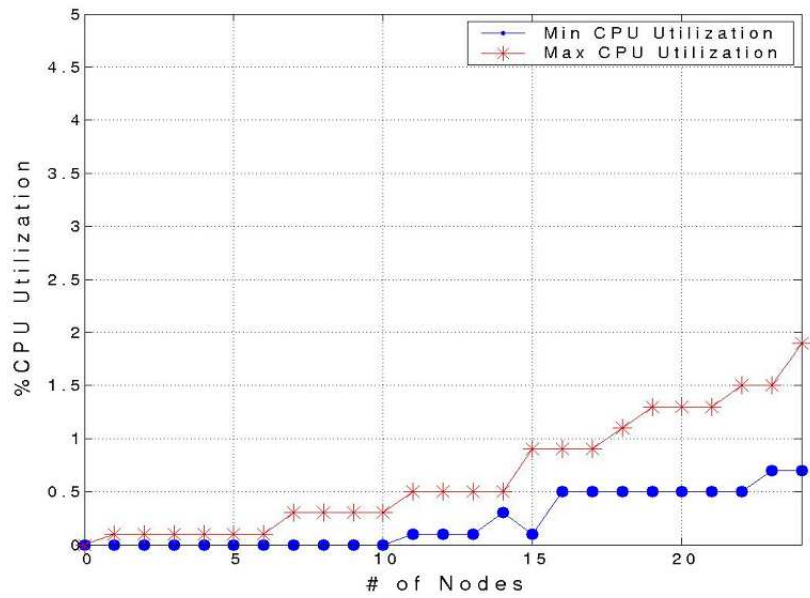


Figure 6.5: %STATMOND NODE 00 CPU usage, Monitoring Interval: 15 secs

6.3.2 Fault-tolerance

In a cluster of nodes, failures are bound to occur. For example, the CPU of a node may fail, or a node may require maintenance and software upgrades. STATMOND has been implemented and tested to work independently of these failures. Though the STATMOND process on a failed node does not participate in the measurement process, the STATMOND process on the other nodes communicate their statistics. Once a failed node is up, it re-joins the multicast group and resumes sending and receiving data. Thus STATMOND is able to tolerate failures in the face of dynamically changing conditions.

6.3.3 Management of Monitored Data

Having a single, centralized repository for dynamic data causes two distinct problems. The first is that the centralized repository for information or control represents a single point of failure for the entire system. Such a failure causes the entire measurement process to stop. Moreover, the data collected over a period of time is neither available to the DRA method nor to other applications. The second problem is that other nodes have to depend on data stored elsewhere and they do not have ready access to dynamic data. The centralized server thus has a overhead to send this data as and when requested by the other nodes of the network. The current implementation of STATMOND hence stores aggregate data

on all nodes in a cluster in standard file formats. These files are accessible to developers, system administrators and to applications that need these data.

6.3.4 Use of Standard Data Formats

STATMOND provides simple options to the user at the command prompt to choose the standard output format. This gives the user of the system flexibility to work with data in ASCII, HTML, or XML formats based on the applications' needs. However, the preferred method is for STATMOND use XML for storing the output. This is because XML parsing is well-defined and widely-implemented, making it possible to retrieve information from XML documents in a variety of environments. Moreover, they are easy to debug compared to other formats such as ASCII. In the case of STATMOND, XML documents are observed to be more verbose than the binary formats and this increases CPU utilization. However, choosing suitable libraries and algorithms can significantly reduce this problem.

6.4 Comparison of STATMOND and Other Tools

Cluster monitoring tools such as NWS, ClusterProbe, PARMON, and GARDMON, typically use daemon processes installed in participating cluster nodes [19] [61] [53] [52]. As a result they can capture data that include both node and net-

work attributes, including CPU loads, memory and swap usage, communication bandwidth, and message round-trip times. STATMOND differs from the above tools in the way the monitoring data is collected, maintained, and sent across the network. The tools above rely on standard Unix utilities such as `vmstat` and `top` to access operating system data. STATMOND uses its own kernel probe to the `/proc` directory, and this increases the speed of access and reduces the level of intrusiveness. Most monitoring subsystems are still based on a centralized daemon that collects information from the distributed agents running on every node of the system. This limits the scalability of the system. The measurement system is also rendered inactive in the event of failure of the central server. Also, such monitoring structures suffer from high latency in data access. STATMOND, on the other hand aggregates data on all the participating nodes. Applications and users can directly pick up the information from a standard output file. A failure of any participating node does not stop the measurement process. While the tools addressed above use traditional communication techniques, STATMOND employs multicast to enhance the scalability. However, it should be noted that the design of each tool is rather different due to the goal and complexity of the monitoring subsystem itself. The focus of NWS is primarily to use the information collected to predict the behavior of large wide area systems. Higher level services that interpret and analyze monitoring data are not the subject of this thesis but

can be a very useful extension to STATMOND. Unlike PARMON, GARDMON, and SCMS, STATMOND does not have a user interface. One of the aims of STATMOND was to integrate the monitoring capability with the user interfaces provided by BigBrother, but this was not achieved.

6.5 Future Work

Though STATMOND is a useful tool for the measurement of system parameters, a number of possibly significant enhancements have been identified that were not implemented within the time frame of this project.

6.5.1 Improve STATMOND's Scalability

One of STATMOND's main features is that it maintains aggregate data on every node on the network. This aggregate data is stored in the form of a table (data structure) in the main memory by the STATMOND process. Each row of the table contains the status and performance metrics corresponding to a particular node on the network. The STATMOND process on each node receives measurement data updates from the respective STATMOND processes on other nodes. STATMOND updates the row in the table that corresponds to the node that has sent the update. Each update is marked by a timestamp: the time at which the update was sent. In

the event several updates are received during one processing interval, STATMOND processes one update at a time.

As the cluster size increases, the frequency of updates to each STATMOND table (data structure) also increase. Often STATMOND receives these updates within a short interval time. This requires more CPU cycles, to process the information received, and this becomes increasingly intrusive as the cluster grows. This limits STATMOND's ability to monitor very large clusters. For instance if a cluster has 500 nodes, it is estimated that STATMOND will use 50% of the CPU time, and this is not an acceptable result for a measurement tool. A method must be designed where STATMOND maintains the most recent data, without receiving too many packets too frequently from all the nodes on the network.

One solution is to use a manageable number of multicast groups (Figure 6.6). This technique has been particularly useful in serving multimedia and Media-on-Demand efficiently across a network [49][58]. Each group has a master node and all master nodes form a different group. Every other node is part of a single group that has a master node. Measurement statistics are forwarded at two levels. The first is at the regular node group level and each node of this group forwards its data through multicast. This information is not available to other groups. The master node at each group then uses muliticast to forward this group level information from to other master nodes. By this time all master nodes have the

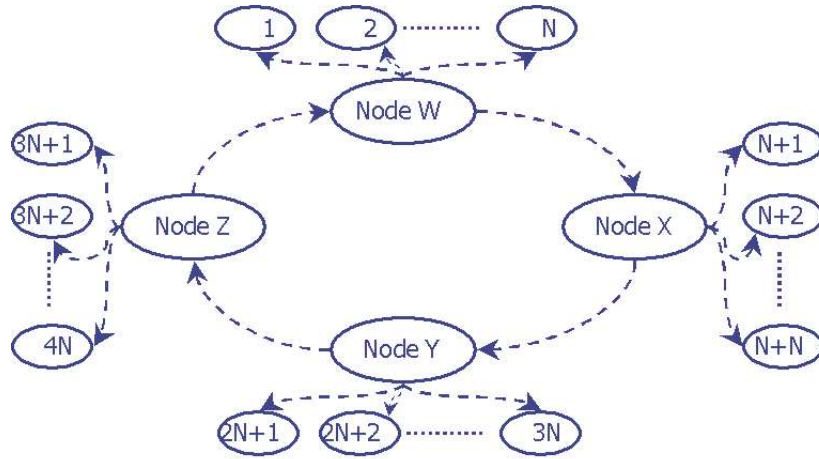


Figure 6.6: Scalability Problem: A Much Better Solution

entire information about all the nodes on the network. The master nodes now transfer this data to the nodes in their respective groups. Hence all nodes get to maintain aggregate statistics.

This method improves scalability because the number of packets received by each node is the size of the subgroup plus as many packets sent by the master node with data from other groups. This number of packets to be processed is significantly smaller than the packets received from all nodes on the network.

6.5.2 Provide a Front-End

A low CPU intensive front end such as a web interface enhances the overall flexibility of the system. Access to the system from any node is made simpler for

the users of the system. Also, several security features and user privileges can be handled by the front end such as a web interface.

This approach has many advantages. The most obvious is that there is no need to write procedural GUI code; instead, it is easier to represent the GUI in specialized languages (HTML and JavaScript). This avoids a lot of expensive and complex single-purpose coding and limits CPU usage. However, there are disadvantages to such a front end. The two most important are:

1. The batch style of interaction that the web enforces where the user fills out a form and clicks a submit button that sends the form contents through a script (such as CGI). The script then runs and the server returns a page of HTML that it generates. This process does not enable a character-by-character or GUI-gesture-by-GUI-gesture I/O through a gateway.
2. The difficulties of managing persistent sessions using a stateless protocol, though these are not exclusively Linux issues. Most front-ends make it easier to keep per-user state on the server. That per-user state can be a problem; it consumes resources, and it has to be timed out, because between transactions there is no way to know that the user is still on the other end of the server.

6.5.3 Extend the Filter Framework

A filter configuration file can make the system very extensible, allowing new filter scripts to be easily added to the system without the need for reprogramming or recompiling. Though this mechanism can deal with some filters, reprogramming is needed for more complicated filters and commands.

6.6 Conclusions

STATMOND is a first step towards an integrated monitoring measurement tool that can facilitate easier dynamic resource allocation on a Linux cluster. As currently implemented the tool provides a simple, text based measurement mechanism that probes, stores, and forwards information across a cluster through the use of multicast. Though this is an improvement over centralized tools, the scalability problem with STATMOND has to be solved to make it more robust and less intrusive. Within the development time frame involved, much was achieved, and the groundwork for further development has been laid.

6.7 Summary

This chapter presents an analysis of the STATMOND tool. It reviews the work carried out and details why STATMOND is more suitable than other tools for

DRA on the LIT cluster. This chapter also addresses further efforts needed to improve STATMOND's scalability and its ease of use.

Bibliography

Bibliography

- [1] Active Measurement Program. <http://amp.nlanr.net/>.
- [2] alog. <http://www.fz-juelich.de/apart/wp3/alog.h>.
- [3] atlas. <http://atlassw1.phy.bnl.gov/cgi-bin/nova-atlas/showMachines.pl>.
- [4] BB4 Technologies Inc., part of Quest Software Inc.
<http://bb4.com/index.html>.
- [5] carnival. <http://www.cs.rochester.edu/u/leblanc/prediction.html>.
- [6] delphi. <http://vibes.cs.uiuc.edu/Project/Delphi/DelphiOverview.htm>.
- [7] dimemas. <http://www.pallas.de/pages/dimemas.htm>.
- [8] dpcl. <http://www.ptools.org/projects/dpcl/>.
- [9] dyninst. <http://www.cs.umd.edu/projects/dyninstAPI/>.
- [10] falcon. <http://www.cc.gatech.edu/systems/projects/FALCON/>.

- [11] The fbi laboratory's combined dna index system program.
<http://www.promega.com/geneticidproc/ussymp6proc/niezgod.htm>.
- [12] gridview. <http://heppc1.uta.edu/kaushik/computing/grid-status/index.html>.
- [13] Iperf. <http://dast.nlanr.net/Projects/Iperf/>.
- [14] jumpshot. <http://www.fz-juelich.de/apart/wp3/clog.h>.
- [15] kojak. <http://www.fz-juelich.de/zam/kojak/>.
- [16] kpi. <http://www.caos.uab.es/kpi.html>.
- [17] mpicl. <http://www.epm.ornl.gov/picl/mpicl.html>.
- [18] mutt. <http://ext.lanl.gov/orgs/cic/cic8/para-dist-team/mutt/muttdoc.html>.
- [19] nws. <http://nws.npaci.edu/NWS/>.
- [20] omis. <http://wwwbode.informatik.tu-muenchen.de/~omis/>.
- [21] pablo. <http://www-pablo.cs.uiuc.edu/>.
- [22] papi. <http://icl.cs.utk.edu/projects/papi/>.
- [23] paradyn. <http://www.cs.wisc.edu/paradyn/>.
- [24] paragraph. <http://www.csar.uiuc.edu/software/paragraph/>.
- [25] pcl. <http://www.fz-juelich.de/zam/PCL/>.

- [26] peridot. <http://wwwbode.in.tum.de/peridot/>.
- [27] picl. <http://www.epm.ornl.gov/picl/picl2.html>.
- [28] Prophecy. <http://prophecy.mcs.anl.gov/>.
- [29] ptools. <http://www.ptools.org/projects/ptr/>.
- [30] rabbit. <http://www.scl.ameslab.gov/Projects/Rabbit/>.
- [31] scheck. <http://cmr.ncsl.nist.gov/scheck/>.
- [32] sddf. <http://vibes.cs.uiuc.edu/Project/SDDF/SDDFOverview.htm>.
- [33] simple. <http://www7.informatik.uni-erlangen.de/tree/IMMD-VII/Research/Groups/>
- [34] Surveyor. <http://www.advanced.org/surveyor/>.
- [35] SvPablo. <http://www-pdsf.nersc.gov/stats/>.
- [36] vampir. <http://www.pallas.de/pages/vampir.htm>.
- [37] Abdallah, C. and N. Alluri and J.D. Birdwell and J. Chiasson and V. Chupryna and Z. Tang and T.W. Wang. A Linear Time Delay Model for Studying Load Balancing Instabilities in Parallel Computations. In *International Journal System Sciences*, November 2002.

- [38] Andrew S. Grimshaw and William A. Wulf and James C. French and Alfred C. Weaver and Paul F. Reynolds Jr. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, 8 1994. citeseer.nj.nec.com/grimshaw94legion.html.
- [39] Birdwell, J.D. and T-W. Wang and M. Rader. The University of Tennessee's new search engine for CODIS. In *6th CODIS Users Conference*, February 2001. Arlington, VA.
- [40] Birdwell, J.D. and T.W. Wang and R.D. Horn and P. Yadav and D.J. Icov. Method of Indexed Storage and Retrieval of Multidimensional Information. In *Tenth SIAM Conference on Parallel Processing for Scientific Computation*, September 2000.
- [41] Brian Tierney and Brian Crowley and Dan Gunter and Mason Holding and Jason Lee and Mary Thompson. A Monitoring Sensor Management System for Grid Environments. In *HPDC*, pages 97–104, 2000. citeseer.nj.nec.com/article/tierney00monitoring.html.
- [42] Brian Tierney and William E. Johnston and Brian Crowley and Gary Hoo and Chris Brooks and Dan Gunter. The NetLogger Methodology for High Performance Distributed Systems Performance Analysis. In *HPDC*, pages 260–267, 1998. citeseer.nj.nec.com/tierney98netlogger.html.

- [43] Dan Gunter and Brian Tierney and Brian Crowley and Mason Holding and Jason Lee. NetLogger: A Toolkit for Distributed System Performance Analysis. In *MASCOTS*, pages 267–273, 2000. citeseer.nj.nec.com/302863.html.
- [44] Debugging Brian Tierney. NetLogger: A Toolkit for Distributed System Performance Tuning and Debugging. citeseer.nj.nec.com/brian97netlogger.html.
- [45] H. Kameda and El-Zoghdy Said Fathy, I. Ryu and J. Li. A Performance Comparison of Dynamic versus Static Load Balancing Policies in a Mainframe. In *Proceedings of the 2000 IEEE Conference on Decision and Control*, pages 1415–1420, December 2000. Sydney, Australia.
- [46] Henri Casanova and Jack Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. Technical Report CS-96-328, Knoxville, TN 37996, USA, 1996. citeseer.nj.nec.com/casanova95netsolve.html.
- [47] Hisao Kameda, Jie Li, Chonggun Kim and Yongbing Zhang. *Optimal Load Balancing in Distributed Computer Systems*. Springer, 1997. Great Britain.
- [48] Ian Foster and Carl Kesselman. GLOBUS: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications*.

- tions and High Performance Computing*, 11(2):115–128, Summer 1997.
citeseer.nj.nec.com/foster96globu.html.
- [49] Marcel Waldvogel and Ramaprabhu Janakiraman. Efficient Media-on-demand over Multiple Multicast Groups. In *Proceedings of Globecom 2001, San Antonio, Texas, USA*, 11.
- [50] Moore, G.E. Cramming more components onto integrated circuits. In *Electronics*, April 1965.
- [51] P. Uthayopas and S. Phatanapherom and T. Angskun and S. Sriprayoonsakul. SCE: A Fully Integrated Software Tool for Beowulf Cluster System, 2001.
citeseer.nj.nec.com/uthayopas01sce.html.
- [52] R. Buyya and B. Koshy and R. Mudlapur. Gardmon: A java-based monitoring tool for gardens non-dedicated cluster computing.
citeseer.nj.nec.com/buyya99gardmon.html.
- [53] Rajkumar Buyya. PARMON: a portable and scalable monitoring system for clusters. *Software Practice and Experience*, 30(7):723–739, 2000.
citeseer.nj.nec.com/buyya00parmon.html.

- [54] R.E. Busby and Jr. M.L.Neilsen and D. Andresen. Enhancing NWS for use in an SNMP managed internetwork. In *Parallel and Distributed Processing Symposium*, 2000.
- [55] Rich Wolski and Neil Spring and Chris Peterson. Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service. 1997. citeseer.nj.nec.com/article/wolski97implementing.html.
- [56] Richard Wolski. Dynamically forecasting network performance using the Network Weather Service. *Cluster Computing*, 1(1):119–132, 1998. citeseer.nj.nec.com/wolski98dynamically.html.
- [57] Richard Wolski and Neil T. Spring and Jim Hayes. Predicting the CPU availability of time-shared Unix systems on the computational grid. *Cluster Computing*, 3(4):293–301, 2000. citeseer.nj.nec.com/article/wolski99predicting.html.
- [58] Sneha Kumar Kasera and Gísli Hjálmtýsson and Donald F. Towsley and James F. Kurose. Scalable reliable multicast using multiple multicast channels. *IEEE/ACM Transactions on Networking*, 8(3):294–310, 2000.
- [59] T. Anderson and D. Culler and D. Patterson. A Case for NOW (Networks of Workstations). 1995. citeseer.nj.nec.com/anderson94case.html.

- [60] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobbs Journal*, 1995.
- [61] Z. Liang and Y. Sun and C. Wang. ClusterProbe: An Open, Flexible and Scalable Cluster Monitoring Tool. citeseer.nj.nec.com/article/liang99clusterprobe.html.

Vita

Krishna Inavolu grew up in Visakhapatnam, Andhra Pradesh, India, where he graduated from Timpany School. He received his undergraduate degree in Electronics and Instrumentation Engineering from Andhra University. He then worked for thirteen months as a software engineer at Infosys Technologies, Bangalore. To pursue higher studies, Krishna enrolled at the University of Tennessee, Knoxville (UTK) in August 2001, and completed the requirements for the Master's Degree in Electrical Engineering in January 2004. During this period, he was a research assistant under Dr. J.D. Birdwell at the Laboratory for Information Technologies (LIT), UTK.

Krishna's work interests lie in the areas of computer networks, high-performance computing, and distributed information systems. After some professional work experience in the industry, Krishna plans to return to school for a Ph.D degree.