GEMS: GOSSIP-ENABLED MONITORING SERVICE FOR HETEROGENEOUS DISTRIBUTED SYSTEMS

By

PIRABHU RAMAN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002

ACKNOWLEDGEMENTS

I wish to thank all the people who have helped in the successful completion of this work. First and foremost, I wish to thank Dr. Alan D. George for his support and guidance throughout this work. He has always inspired and encouraged me to perform better. I also extend my thanks to my team leader, Matt Radlinski, for reviewing and helping me with the slides and paper in spite of being busy with his own work.

I would also like to thank the members of the HCS Lab, Burt, Julie, Kyu, Adam and especially my friend Raj, for their reviews and suggestions. Raj has been a great source of inspiration and has helped me most patiently with even the smallest of details. He has helped in making this a much better work. I would also like to thank Sridhar, who was of immense help during the earlier part of this work. I also extend thanks to my friends Charles, Karthik, Aravind, Venkat, Nand, Ninja, Rama and Hari, who made this MS livelier.

I thank my beloved parents for being there and believing in me for all the years. I would also like to thank my brother, who has been more of a friend than a brother and who has been an endless source of encouragement in all my endeavors.

Finally, I thank all the sponsors of this work. This work was supported by Sandia National Labs on contract LG-9271, and by equipment grants from Nortel Networks, Intel, and Dell.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

GEMS: GOSSIP-ENABLED MONITORING SERVICE FOR HETEROGENEOUS DISTRIBUTED SYSTEMS

By

Pirabhu Raman

December 2002

Chair: Alan D. George
Major Department: Electrical and Computer Engineering

Gossip protocols provide a scalable means for detecting failures in heterogeneous distributed systems in an asynchronous manner without the limits associated with group communication. In this work, we discuss the development and features of a hierarchical Gossip-Enabled Monitoring Service (GEMS), which extends the gossip-style failure detection service to support resource monitoring. By dividing the system into groups of nodes and layers of communication, the GEMS paradigm scales well. Easily extensible, GEMS incorporates facilities for distributing arbitrary system and application-specific data. In this work we present experiments and analytical projections demonstrating fast response times and low resource utilization requirements, making GEMS a superior solution for resource monitoring issues in distributed computing. Also, we demonstrate the utility of GEMS through the development of a simple dynamic load balancing service for which GEMS forms the information base.

CHAPTER 1
INTRODUCTION

With the advent of high-speed networks, heterogeneous clusters built from commercial off-the-shelf components offer a good price/performance ratio and are an attractive alternative to massive parallel processing computers. However, heterogeneous, high-performance clusters have to overcome many challenges in order to achieve peak performance--among them, failure detection and resource monitoring. Failures plague clusters designed from stand-alone workstations and personal computers, and such systems require failure detection services to monitor nodes and report failures, allowing self-healing and check-pointing applications to restart dead processes.

With the increased availability of advanced computational power, a need exists for detecting and monitoring idle resources among identical nodes in a homogeneous system in order to reduce computation and response times. The need is even greater in a heterogeneous environment where resources vary in quantity and quality from one node to another. Resource monitoring systems serve as an information source for performance information and the location and usage of resources. Time-critical services and long running applications with process migration capabilities can use the monitored information to distribute processes and tasks. Resource monitoring information can also be used for initiating diagnostics on a node, adding new nodes and resources to services as they become available, and reconfiguring the system; for example, redirecting packets to bypass overloaded routers. Thus, resource monitoring provides a critical low-level

service for load balancing and scheduling middleware services, apart from providing a single system image to users and administrators.

In this work, we develop a resource monitoring service by piggybacking resource monitoring data on a gossip-style failure detection service. In Sistla et al. [1-2], the authors demonstrate a highly scalable gossip-style failure detection service, capable of supporting hundreds of nodes with low resource utilization. The service also employs distributed consensus for fast and reliable failure detection with reaction times in the milliseconds range, even for larger systems. Hence, the use of gossip as the carrier for the resource monitoring service data addresses the challenges of clustering, failure detection and resource monitoring, and reduces overhead as well. Apart from the dissemination of resource data, the GEMS Application Programming Interface (API) allows users to share application data and introduce user-defined aggregators into the service for easy expansion. Finally, as a case study, we develop a simple load balancing service to demonstrate the use of GEMS as an information service for load balancing and scheduling middleware services.

The rest of the thesis is organized as follows. The next chapter discusses other resource monitoring services and related work. Chapter 3 describes the gossip-style failure detection service, while the resource monitoring service is developed in Chapter 4. Resource utilization experiments and performance results, demonstrating the scalability of the service, follow in Chapter 5. In Chapter 6 we develop the load balancing case study to demonstrate the application of the resource monitoring service. Finally, conclusions and directions for future research are discussed in Chapter 7.

CHAPTER 2
RELATED WORK

Though several resource-monitoring services have been developed for heterogeneous clusters of workstations, few meet the scalability and extensibility requirements of heterogeneous clusters. These include services such as the Network Weather Service, Load Leveler, Cluster Probe, Parmon and Astrolabe. We briefly discuss the architecture of these services and present some of their shortcomings that prompted us to develop GEMS.

The University of California at Santa Barbara developed the Network Weather Service (NWS) [3-4], a resource monitoring and forecasting service for clusters of workstations. NWS predicts the performance of a particular node using a time series of measurements. The service is comprised of three parts: the name server, the persistent state processes and the sensors. Sensors gather performance measurements and report them to the local persistent state processes, which in turn store the measurements on a permanent medium. The measurements are used to generate forecasts by modules called predictors.

NWS includes a small API of two functions for retrieving the performance forecasts. Applications initially call the *InitForecaster* function which initializes the predictor with a recent history of measurements. When applications are ready to receive forecast data, they call the *RequestForecasts* function to send the request message. After updating with all the measurements that have been made since the last call to either of these two functions, the predictor generates the forecasts. Thus, applications that call the

RequestForecasts function infrequently will experience long delays due to the time spent in updating the predictors. Another limitation is resilience, since a failure of the persistent state process results in loss of data as the measured data are no longer stored in the permanent medium.

Load leveler [5], a batch job scheduling application developed by IBM, has a central manager that stores monitored information from all nodes in the system. The service suffers from a single point of failure, and poor scalability due to the bottleneck at the central manager. Cluster Probe [6], a Java-based resource monitoring tool developed at the University of Hong Kong, has a central manager and many proxies to improve scalability, where a subset of the nodes report to the proxy and the proxies in turn report to the central manager. The tool suffers from a single point of failure at the central manager and is not resilient against proxy crashes.

Parmon [7] is another resource monitoring service developed at the Centre for Development of Advanced Computing (CDAC). Parmon monitors an extensive array of system parameters using client server architecture and has a graphical user interface (GUI) developed in Java. The Parmon server should be active on all the nodes that are monitored. The GUI based Parmon client receives monitoring requests from the user and polls the Parmon server on each machine requested by the client. Thus, the service as such is not scalable beyond hundreds of nodes and the requests along with response messages introduce significant network overhead.

In Astrolabe [8], a scalable and secure resource location service developed at Cornell University, the data aggregation is done using gossip-style communication. The service uses simple timeouts for failure detection by the management information base

(MIB), which stores monitored resource data. The timeout mechanism is unreliable and increases the probability of erroneous failure detection. A reliance on wall-clock time, which is used in the service as timestamps for differentiating between update messages, will lead to synchronization problems and increased resource utilization. The authors provide a simulative analysis of the speed of propagation of updates in the face of failures but fail to offer a scalability analysis in terms of resource utilization.

In summary, a distributed monitoring service which periodically disseminates the monitored data is needed for faster response times. The service needs to be flexible, with options for monitoring both individual nodes and groups of nodes, to support system administration. Finally, large distributed applications with long run times require the service to be scalable and fault-tolerant.

CHAPTER 3
GOSSIP-STYLE FAILURE DETECTION SERVICE

Gossiping [9-10] is a scalable failure detection technique for heterogeneous clusters, employing a simple point-to-point communication scheme in which each node sends a message to a randomly selected node at a fixed time interval. When a node, say "node 1," fails to receive liveness information about another node, say "node 2," for a fixed period of time, either directly as a gossip message from node 2 or from other nodes in the system, node 1 suspects that node 2 may have failed. When all the nodes suspect the failure of node 2, consensus is reached and node 2 is declared dead. Thus, the gossip-style failure detection service does not depend on any single node or message for its operation and hence is very resilient. Also, the point-to-point communication scheme of gossip makes minimal assumptions about the characteristics of the network and is more scalable and portable than group communication schemes [11-12].

Three key parameters, the gossip interval, cleanup time and consensus time determine the performance of the gossip-style failure detection service. The gossip interval, or T_{gossip}, is the time period between two consecutive gossip messages. If an active node has not heard from a remote node for an interval of $T_{cleanup}$, the cleanup time, then the active node suspects the remote node may have failed. These two are input parameters for a particular gossip-based failure detection system. The system responds quickly to changes in node liveness when both these values are relatively small, but at the price of increased resource utilization. Finally, consensus time, or $T_{consensus}$, is the time interval after which consensus is reached about a failed node.

3.1 Consensus Algorithm

Each node maintains three data structures: a gossip list, a suspect vector and a suspect matrix. The three structures together represent the perceived state of each node within a system. The gossip list is a vector of size n, where n is the number of nodes in the system. Element j of the gossip list at node i contains the number of T_{gossip} intervals, also referred to as heartbeat values, since node i has received any gossip information from node j. If the value in element j is greater than $T_{cleanup}$, then the corresponding element of the suspect vector is set to "1" indicating the node is suspected to have failed, otherwise it remains "0" indicating a healthy node. The suspect vectors of all n nodes together form a suspect matrix of size $n \times n$. Each node sends a gossip message containing its gossip list and the suspect matrix to a randomly selected node. On receipt of a gossip message, the state view of a node is updated by merging the data structures as explained in Ranganathan et al. [13].

Node 1 gossip list

Node	Heart beat
1	0
2	***3***
3	***4***

+

Node 2 gossip list

Node	Heart beat
1	3
2	***0***
3	***2***

=

Node 1 gossip list
After merging

Node	Heart beat
1	0
2	***0***
3	***2***

Figure 3-1. Gossip list update

3.1.1 Gossip List Update

Gossip lists are updated by comparing the heartbeat values maintained at the node with the heartbeat values in the received message. Figure 3-1 illustrates a gossip list

update in a three-node system. Here, node 1 receives a gossip message from node 2 and merges the received list with its own. First, the heartbeat values corresponding to node 1 in the two lists are compared. Node 1 knows it is still alive, so its heartbeat value is always "0," and cannot be overwritten. Next, the heartbeat values corresponding to node 2 are compared. Since the message is received from node 2, the heartbeat value corresponding to node 2 is reset to zero, indicating that node 1 has just heard from node 2. Finally, the heartbeat values corresponding to node 3 are compared. Node 2 last heard from node 3 only two gossip intervals ago, whereas node 1 heard from node 3 four gossip intervals ago. So, node 1 updates its gossip list to reflect the fact that node 3 was in fact alive two gossip intervals ago.

3.1.2 Suspect Matrix Update

The suspect matrix is an $n \times n$ matrix with the rows and columns of the matrix representing the nodes in the system. Whenever a node i suspects node j, the element (i, j) of the suspect matrix is set to "1." When all the nodes in the system except the suspected nodes contain a "1" in column j of the suspect matrix then consensus is reached on the failure of node j. Figure 3-2 illustrates the update of the suspect matrix in a three-node system. In the figure, both node 1 and node 2 separately suspect node 3 and no other nodes suspect node 1 and 2. On receipt of a gossip message from node 2, node 1 updates its suspect matrix and determines that node 2 also suspects node 3. Hence, a consensus is reached on the state of node 3 at node 1. Node 1 changes its livelist entry for node 3 to "0," indicating the node is dead, and broadcasts the consensus information to the other nodes in the system.

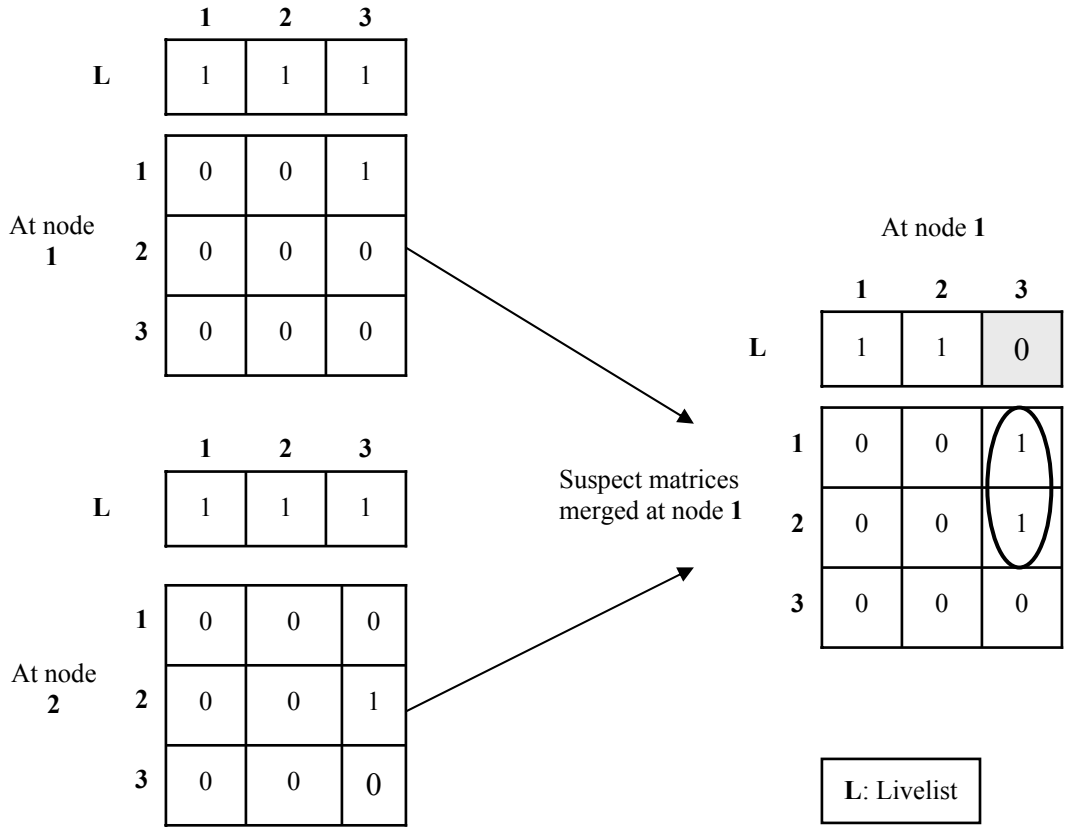


Figure 3-2. Consensus in a three-node system

3.2 Layered Gossiping

The distributed consensus algorithm works efficiently for small systems. However, the size of the suspect matrix and gossip list grows dramatically with system size leading to an increase in communication and processing overhead. Additionally, the minimum value allowed for $T_{cleanup}$ increases, increasing failure detection time as well.

These problems are addressed by dividing the system into groups or clusters in a hierarchy of two or more layers, typically chosen to fit the network topology. In this setup, each node gossips only within its local group, and higher layers handle gossiping between groups. For instance, in a two-layered system nodes in each group take turns sending messages to other groups through the upper-layer communication channel.

When consensus is reached within a lower-layer group, this information is broadcast to all the nodes in the system. An example of a two-layered system is shown in Figure 3-3. Layering groups of nodes in this way also facilitates increased heterogeneity in the system by grouping similar nodes together.

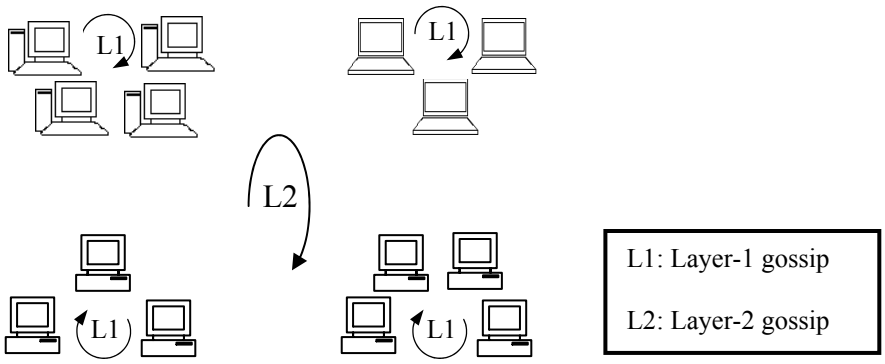


Figure 3-3. An example of a two-layered system

CHAPTER 4
RESOURCE MONITORING SERVICE

GEMS extends the gossip-style failure detection protocol to support resource monitoring. In addition to using the gossip-style failure detection service as a carrier to reduce overhead, the service also uses the gossip heartbeat mechanism to maintain data consistency. In this chapter, we discuss the basic architecture of our resource monitoring service, its components, and the mechanism by which our service guarantees data consistency.

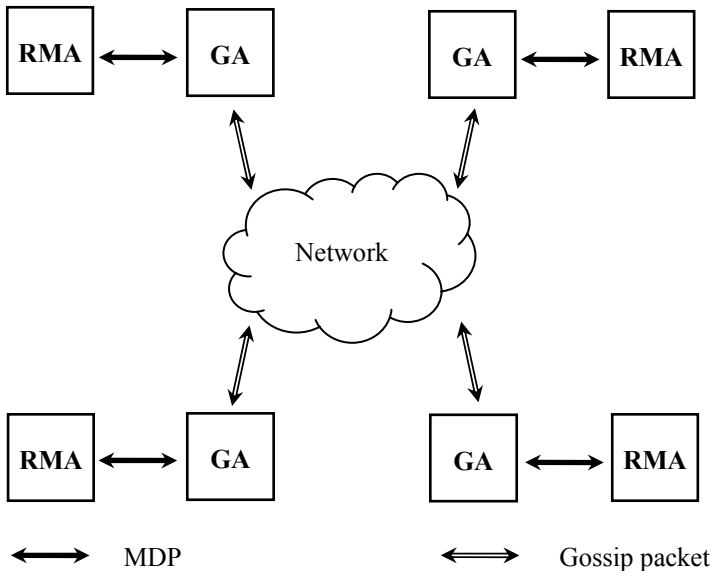


Figure 4-1. MDP exchange in the resource monitoring service

The resource monitoring service is made up of two main components: the Resource Monitoring Agent (RMA) and the Gossip Agent (GA). The RMA and GA should be active on each node that forms part of the resource monitoring service. Figure 4-1 shows the exchange of MDPs in the resource monitoring service. The RMA gathers

monitored data from the system and forms a Monitored Data Packet (MDP), which is forwarded to the GA. The GA piggybacks the MDP onto the failure detection service's gossip messages and receives MDPs from other nodes and forwards them to the RMA. The RMA initially registers with the gossip agent, in the absence of which the gossip agent ignores any incoming monitor data packets and does not forward them to the RMA.

Forming the basic block of the resource monitoring service, the RMA is composed of the sensors, the communication interface and the Application Programming Interface (API). Figure 4-2 shows the various components of the resource monitoring service with the RMA as the middleware between the hardware resources and user-level applications. The communication interface of the RMA receives the system parameters measured by the sensors and the queries from the API. The API provides a set of functions for the dissemination and retrieval of data by applications and middleware services.

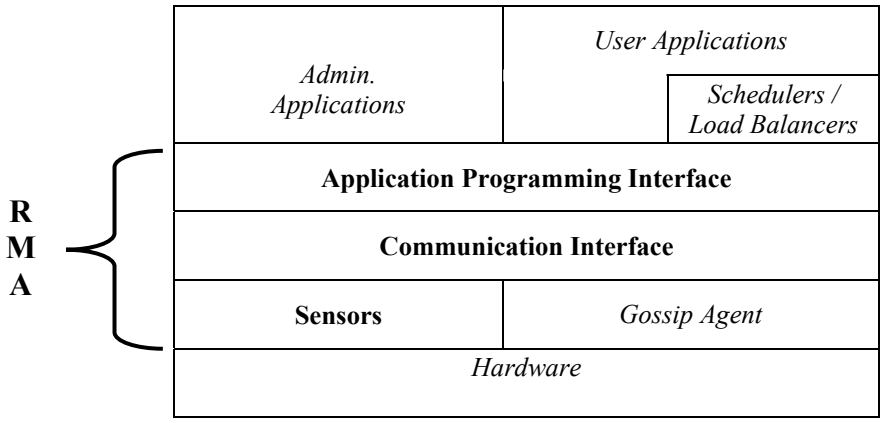


Figure 4-2. Components of the resource monitoring agent

4.1 Sensors

The sensors interact with the hardware and applications to gather resource data, which forms the MIB. GEMS has two types of sensors: built-in sensors and user-defined

sensors. Built-in sensors are provided with the service, and measure the following parameters:

- *Load average* – 1/5/15 min CPU load average
- *Memory free* – Available physical memory
- *Network utilization* – Bandwidth utilization per node
- *Disk utilization* – Number of blocks read and written
- *Swap free* – Available free swap space
- *Paging activity* – Number of pages swapped
- *Num. processes* – Number of processes waiting for run time
- *Virt. memory* – Amount of virtual memory used
- *Num. switches* – Number of context switches per second

The user-defined sensors measure new system and application-specific parameters, which are useful for monitoring resources that are not supported by the built-in sensors. For example, a new user-defined sensor that measures the round-trip latency between two nodes could be developed with minimal effort using the ping networking service. The measured latency parameter can be disseminated through the resource monitoring service using the API function calls. Since there is no limit associated with the amount of user data that each application can send, an RMA might receive several different types of user data from various applications. These user data are identified using unique IDs, which are numerical tags assigned by the RMA.

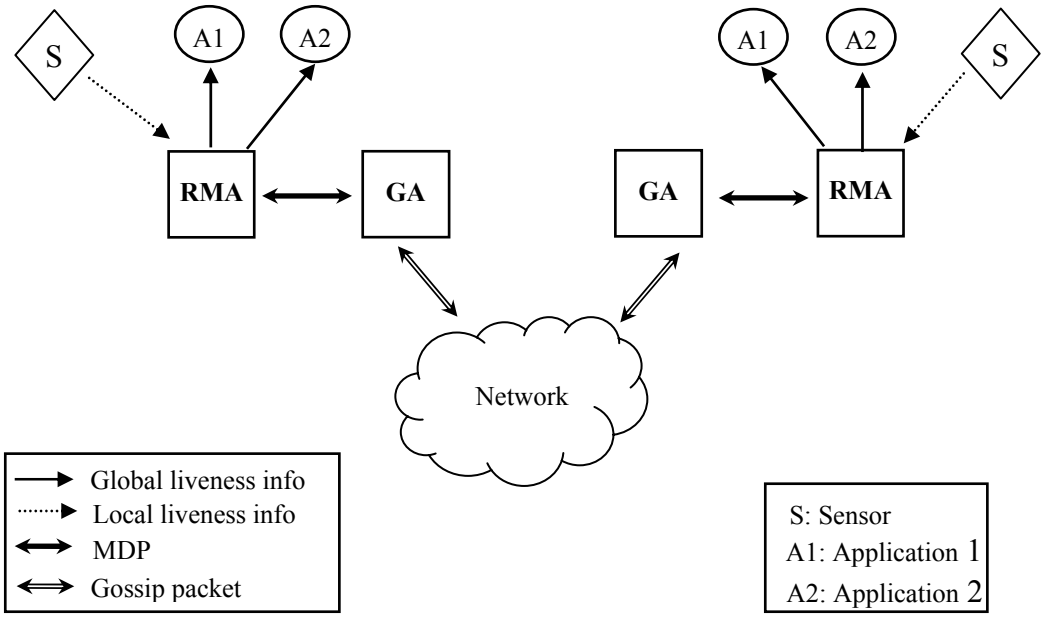


Figure 4-3. Example scenario describing the user data ID

Figure 4-3 shows an example scenario illustrating the usefulness of assigning IDs for identification of user data. Consider a user-defined sensor developed for failure detection of processes. Such a sensor needs to be implemented because gossip currently supports only node-level failure detection, but failure detection for processes is a simple extension. An application with its processes distributed across the nodes can contact the RMA using the API and obtain a globally unique ID *i*. The application process then registers itself with the failure detection sensor providing it the local Process ID (PID) and the application's assigned ID *i*. Other instances of the same application, running on various nodes, should do the same with their local sensors using their local PID and the same globally unique ID *i*. The sensor can be designed to check the processes periodically for liveness and disseminate the monitored information through the resource monitoring service using ID *i*. Eventually, the applications can contact the RMA periodically using the ID *i* and know which of its processes are alive. In Figure 4-3, A1 and A2 are two applications with processes distributed on more than one node. S, the

process failure detection sensor, checks the local processes of applications A1 and A2 and reports their status to the RMA. A1 and A2 might be assigned IDs "1" and "2" respectively by the RMA. Thus, any process in any node with an active RMA can use ID "1" to determine the liveness of all processes that belong to application A1.

4.2 Communication Interface

The communication interface, which forms the heart of the Resource Monitoring Agent, receives queries from applications through the API, local monitored data from the sensors and external monitored data from the gossip agent. The communication interface is responsible for generating the Monitored Data Packet, updating the MIB with newly received data and executing the aggregation functions to generate the aggregate MIB.

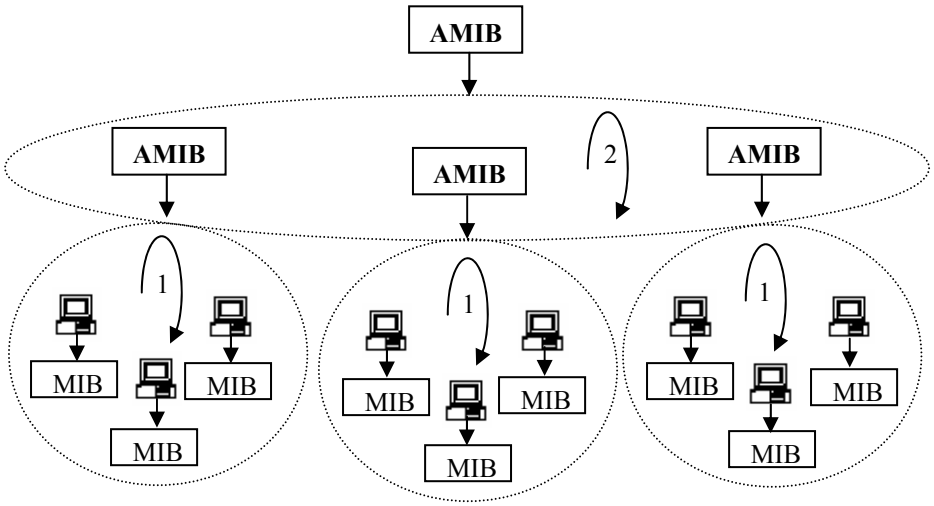


Figure 4-4. Structure of the resource monitoring service

Aggregation functions use the monitored data of all nodes in the group, to generate a single aggregate MIB (AMIB) representing the whole group. Figure 4-4 shows the structure of a two-layered resource monitoring service having three groups, each with three nodes. Sensors read the system parameters that form the local MIB and the local MIB of all nodes in a group together form an AMIB. The local MIB is

exchanged within the group through layer-1 gossip communication whereas the AMIB is exchanged between groups through higher layers of gossip communication. The aggregation of monitored data reduces resource consumption while providing information sufficient for locating resources in the system. While the figure shows the communication for only two layers, similar aggregation may be performed for layers higher than layer-2. GEMS currently supports aggregation for only two layers, but extensions for multiple layers are planned.

4.2.1 Aggregation Functions

Aggregation functions operate on the data monitored by sensors to generate an AMIB. The applications can choose a particular aggregation function to suit the data type on which they operate. Some basic aggregation functions such as the mean, median, maximum, minimum, summation, and Boolean aggregation functions are provided with the service. The functions are identified by specific IDs similar to user data. Figure 4-5 illustrates a situation in which these functions generate an aggregate MIB for a two-node group. Here, three parameters are aggregated using different aggregation functions. When the knowledge concerning the availability of a resource is required, the presence or absence of the resource can be represented by the use of a single bit. Thus, a Boolean aggregation that performs an 'OR' operation is used in this case. In the case of load average perhaps only the average load of the whole group would make sense and hence a mean aggregation function is used in such cases.

4.2.2 User-Defined Aggregation Functions

The aggregation functions provided with the service are limited and applications might require new modes of aggregation that are better suited for their data. Features have been added for introducing new user-defined aggregation functions into the system.

Like the built-in aggregation functions, IDs are used for identification of user-defined aggregation functions. These aggregation functions are dynamically loaded into the service without the need for the recompilation or restart of the service.

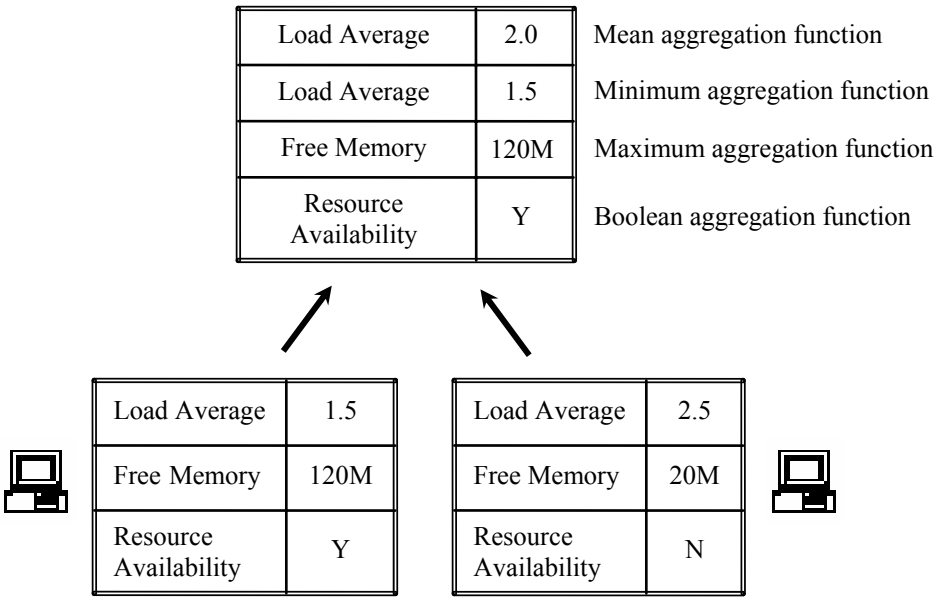


Figure 4-5. Illustration of the built-in aggregation functions

4.2.3 Monitor Data Packet (MDP)

The MDP contains the monitored information, which includes data from the sensors for individual nodes in the group and the aggregate data of the groups. Figure 4-6 shows the layout of layer-1 and layer-2 monitored data packets in a two-layered system. Here, the layer-1 packet contains the aggregate data of only the layer-2 groups whereas in a multi-layered system, the layer-1 packet will have the aggregate data of all the upper layer groups higher than layer-1. Thus, size of the layer-1 packet depends on the number of nodes/groups in each layer while that of layer-*n* packet depends only on the number of groups in layer-*n* and higher. Layer-1 packets, therefore, are the largest packets in the system.

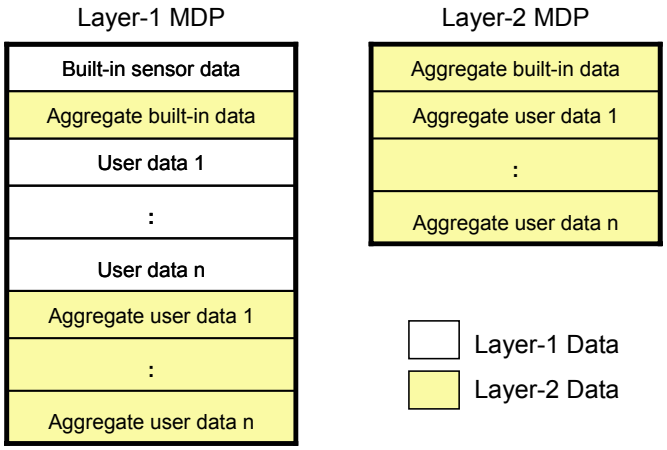


Figure 4-6. Monitor data packet

4.2.4 Data Consistency

Monitor Data Packets contain both new and stale data, which are updated using the gossip heartbeat values similar to the gossip list update. Whenever a new MDP is received, the communication interface merges the received contents with data previously present in the local RMA. Figure 4-7 illustrates an MIB update that occurs when node 1 receives an MDP from node 2 for a metric called "load." The load metrics with lower heartbeat values in the incoming MDP of node 2 are updated into the MIB of node 1. In the figure, the heartbeat values corresponding to node 2 and node 3 are lower in the received MDP than the current MIB values at node 1, so the load values of these nodes are updated. The heartbeat values corresponding to node 4 and node 1 are lower at node 1, and hence the previous load values for node 4 and node 1 are retained.

4.3 Application Programming Interface (API)

The Application Programming Interface consists of a set of easy-to-use dynamic library functions. The API forms the principal mode of communication between the application and the Resource Monitoring Agent. The functions are broadly categorized into initialization, control and update functions.

Node 1's data

Node	Heart Beat	Load
1	0	2.0
2	***10***	***6.0***
3	***90***	***4.0***
4	20	4.0

+

Incoming data from node 2

Node	Heart Beat	Load
1	40	3.0
2	***0***	***3.0***
3	***20***	***2.0***
4	40	1.0

=

Node 1's new data

Node	Heart Beat	Load
1	0	2.0
2	***0***	***3.0***
3	***20***	***2.0***
4	20	4.0

Figure 4-7. Updating monitor data packets

4.3.1 Initialization Functions

The initialization functions are used for registering the RMA with the gossip agent as well as procuring IDs for the user data and aggregation functions. Table 4-1 shows the various monitor initialization functions, their operation and their return values.

Table 4-1. Initialization functions

API function name	Operation	Return values
gems_init	**gems_init**	*Success/failure*
gems_userdata_init	**gems_userdata_init**	*New ID for user data*
gems_aggfn_init	**gems_aggfn_init**	*New ID for aggregation functions*

4.3.2 Control Functions

The control functions enable and disable the operation of RMA as well as the dissemination of monitored data. Table 4-2 shows the various monitor control functions, their operation, and their return values.

Table 4-2. Control functions

API function name	Operation	Return values
gems_start	Starts RMA	*Success/failure*
gems_end	Stops MDP dissemination	*Success/failure*
gems_userdata_stopall	Stops dissemination of all user data	*Success/failure*
gems_userdata_stop	Stops dissemination of data identified by ID	*Success/failure*

4.3.3 Update Functions

These functions update applications with built-in sensor data and user data from the RMA. The users can choose to receive all the application data disseminated by GEMS using the *gems_update_w_userdata* function or selectively receive data of specific applications using the *gems_update_w_nuserdata* function. In the latter function, data of specific applications are received by providing the corresponding user data IDs, with '*n*' referring to the number of such data requested by the client. Finally, the functions *gems_senduserdata* and *gems_recvuserdata* are used for dissemination of user data using GEMS. Table 4-3 shows the various monitor update functions, their operation and their return values.

Table 4-3. Update functions

API function name	Operation	Return values
gems_update	RMA query function	*Built-in sensor data*
gems_update_w_userdata	RMA query function	*Built-in sensor and user data*
gems_update_w_nuserdata	RMA query function	*Built-in sensor and 'n' user data*
gems_senduserdata	Sends user data to RMA	*Success/failure*
gems_recvuserdata	Receives user data from RMA	*User data of nodes and aggregate data of group*

4.4 Steps in User Data Dissemination

The dissemination of user data involves the following steps: procuring an ID for the user data, procuring an ID for the aggregation function if the data employs a user-defined aggregation function, and the selection of an aggregation function. The application has to first confirm whether an RMA is active at the node and, if not, spawn a new RMA process. If a new user-defined aggregation function is required, then a request for a unique ID is sent to the RMA, along with the filename containing the aggregation function. The RMA assigns the unique ID to the new function, after dynamically loading it into the service and propagates it throughout the system. The application then requests an ID for the user's data, and upon receipt the application sends the user data, the data ID, and the aggregation function ID to the RMA. Henceforth, the application disseminates the data using this ID, until such time that it notifies the RMA to discontinue service for this piece of user data. Figure 4-8 shows a simple flow chart detailing the procedure for the dissemination of user data, while the pseudo-code with appropriate API functions is shown in Figure 4-9.

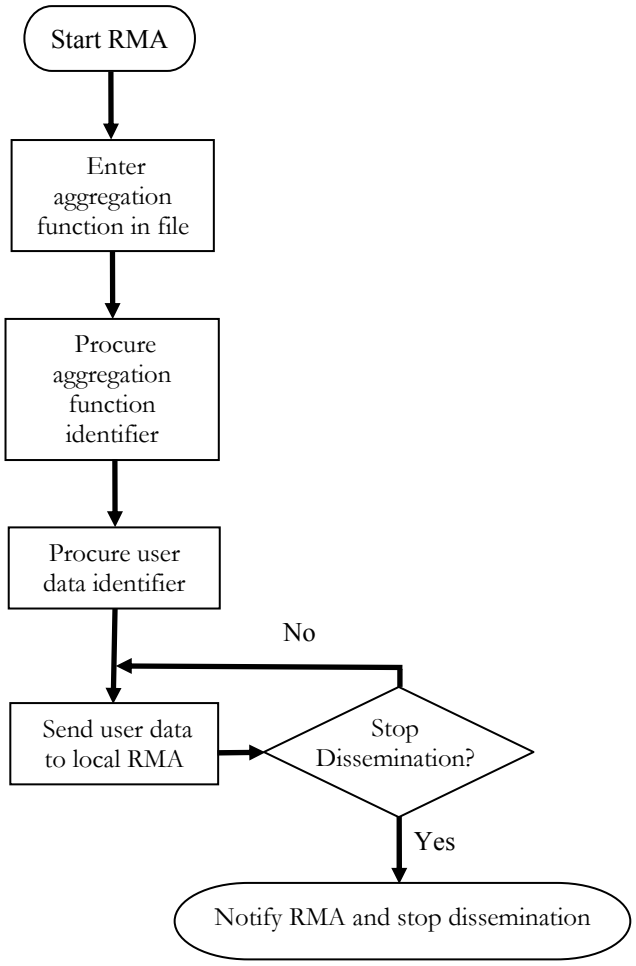


Figure 4-8. Flow chart for dissemination of user data

```
1.  userdata_id = gems_userdata_init( );                    //get IDs from RMA --- only one
2.  aggfn_id = gems_aggfn_init( );                          //of the nodes should do this
3.  while (1)
4.  {
5.          data = sensor ( );                              //Update my local data
6.          gems_senduserdata (userdata_id, aggfn_id, data); //send my local data
7.          gems_recvuserdata (userdata_ id);               //receive data of all hosts from RMA
8.          sleep (1);
9.  }
10. gems_userdata_stop (userdata_id);                       //stop dissemination
```

Figure 4-9. Pseudo-code for dissemination of user data

CHAPTER 5

EXPERIMENTS AND RESULTS

We conducted a series of experiments involving the measurement of network and CPU utilization to verify the scalability of the service. The experiments were conducted on a PC cluster of 240 nodes running the Linux RedHat v7.1 or v7.2 operating system and kernel version 2.4. All the experiments illustrated below used Fast/Gigabit Ethernet for communication with UDP as the network transport protocol.

The primary focus of the experiments was to determine the configuration that yields minimum resource utilization for a given system size, as well as ascertaining the scalability of the service. In a related work, Sistla et al. [1-2] performed an experimental analysis of flat and layered gossip-style failure detection schemes. Based on their experimental results and analytical models, the authors deduced the optimum group size for achieving minimum resource utilization. Experimental observation shows that the optimum group size for the gossip service piggybacked with MDP matches the optimum group size deduced in this previous work. Experiments also show that the service has good scalability in resource utilization versus system size.

5.1 Optimum Group Size

Consider a two-layered system of g groups each containing m nodes, where $N = m \times g$ is the system size. Each node sends a first-layer gossip message every T_{gossip} seconds while it sends a second-layer gossip message once every $m\ T_{gossip}$ seconds. The

network bandwidth per node consumed by the MDP payload B_{MDP} in bytes/sec is given by

$$B_{MDP} = \frac{(m \times d) + (g \times d)}{T_{gossip}} + \frac{(g \times d)}{m \times T_{gossip}} \quad (1)$$

where d is the size of data monitored at each node. From Sistla et al. [2], the network bandwidth per node consumed by the two-layered gossip-style failure detection service, B_{gossip}, in bytes/sec is

$$B_{gossip} = \frac{44 + (m+1) \times \left(\left\lceil \frac{m}{8} \right\rceil + 1 \right) + (g+1) \times \left(\left\lceil \frac{g}{8} \right\rceil + 1 \right)}{T_{gossip}} + \frac{45 + (g+1) \times \left(\left\lceil \frac{g}{8} \right\rceil + 1 \right)}{m \times T_{gossip}} \quad (2)$$

Therefore, the combined network bandwidth per node consumed by the gossip-style failure detection service messages and the monitoring service payload, B_{GEMS}, in bytes/sec is given by

$$B_{GEMS} = B_{gossip} + B_{MDP} \quad (3)$$

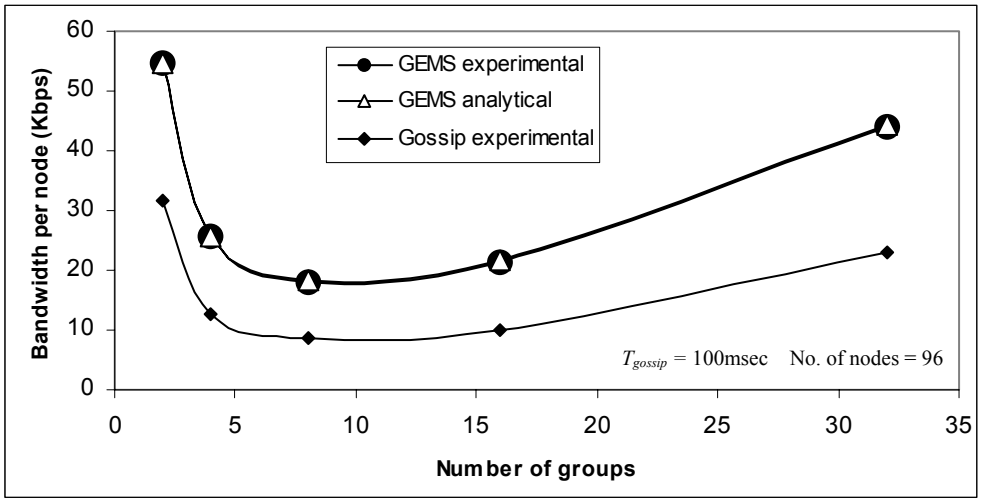
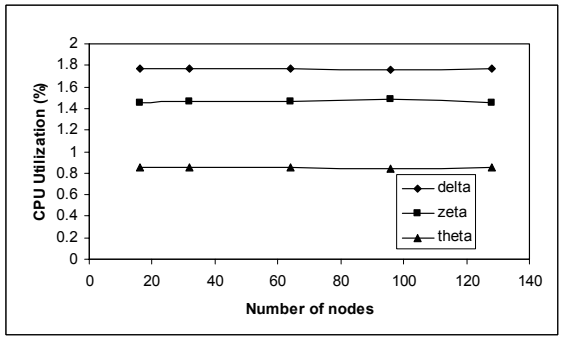


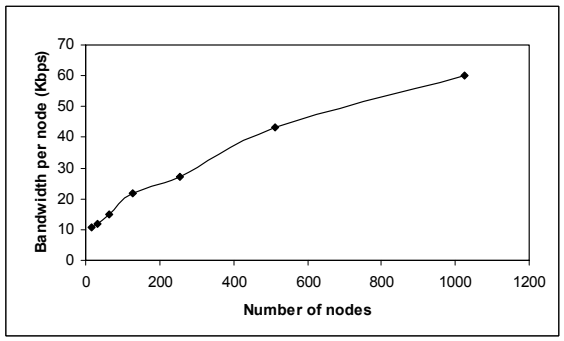
Figure 5-1. Network utilization in a two-layered system for various group sizes

Figure 5-1 shows the network bandwidth utilization per node of the gossip-style failure detection service and GEMS. Apparent from the figure, the optimum group size for a 96-node system is eight. As previously mentioned, Sistla et al. [1-2] proposed a simple heuristic for determining the optimum group size for a given system size. The optimum group size is approximately equal to the square root of system size, rounded to a multiple of eight due to the byte-oriented structure of gossip packets. Thus, for a system size less than 128, eight is the optimum group size while 16 is the optimum group size for system size in the range 128 to 512. The upper curve shows the measured bandwidth consumption of the GEMS service. The analytical values were plotted using Eq. 3. It can be observed that the optimum group size of GEMS is eight, the same as that for the failure detection service. Thus, for a system of N nodes the optimum group size for achieving minimum resource utilization with GEMS is approximately equal to $\sqrt{N}$.



T_{gossip} = 100 ms; No. of groups = $\lceil \sqrt{N} \rceil$

a) CPU utilization



T_{gossip} = 100 ms; No. of groups = $\lceil \sqrt{N} \rceil$

b) Network utilization

Figure 5-2. Resource utilization vs. system size

5.2 Resource Utilization

Figure 5-2 shows the network and CPU utilization for various system sizes using their corresponding optimum group size. The network utilization values for systems greater

than 128 nodes are analytically projected using the formula given in Eq. 3. The CPU utilization experiments were performed on a variety of machines in the cluster whose specifications follow:

- ✓ Delta node – 600MHz Intel Pentium-III processor with internal 512KB L2 cache
- ✓ Zeta node – 733MHz Intel Pentium-III with integrated 256KB L2 cache
- ✓ Theta node – 1333MHz AMD Athlon processor with integrated 256KB L2 cache

From the graphs it can be inferred that the resource utilization increases at a slow rate with system size. For a system size of 1024 nodes, the bandwidth consumption is less than 60 Kbps, while the CPU utilization for a system of 128 nodes is less than 2%. From Figure 5-2(b) it can be observed that the network utilization increases linearly with system size. CPU utilization will follow the same trend. Switching to multiple layers for higher system sizes can reduce the linear increase in resource utilization. In fact, in recent related work [14], the gossip-style failure detection service has been enhanced to support an arbitrary number of layers and these enhancements will yield increased scalability with low resource utilization for GEMS.

CHAPTER 6
CASE STUDY

A critical application of a resource monitoring service is to provide information for load balancing and scheduling middleware services. To demonstrate a practical application of GEMS, we have developed a simple load balancing service that uses GEMS as its information base.

Load balancing in a distributed system is the process of sharing resources by transparently distributing the system workload. Large applications such as ray tracing and fluid dynamics simulations may have long execution times depending on their input data and level of fidelity. These applications can be optimized for distributed computing by parallelizing the applications and taking advantage of the inherent concurrency in them. Still, the dynamic nature of these applications produces unbalanced workloads on the processors during their execution. Dynamic load balancing will distribute tasks in the system proportional to the load on each node, thereby yielding higher throughput and more efficient resource utilization.

Dynamic load balancing systems are characterized by information policy, location policy and transfer policy [15-16]. Information policy determines what information about the load states in the system is needed, and how such information is collected and disseminated. Transfer policy determines whether a task should be transferred depending upon the load of the target node. Location policy determines the appropriate target host for the task.

Information Policy. GEMS forms the information service for our load balancing application. When using a flat gossiping scheme, GEMS disseminates information for all the nodes in the system. With the layered gossiping scheme, GEMS disseminates individual node information within the group and aggregate information between groups. Finally, GEMS employs periodic dissemination of monitored data and hence any application that needs information can query the local GEMS agent and receive current monitored data.

Transfer Policy. Task transfers are initiated when the host load is greater than a threshold value [16]. The threshold value used here is the average load of all the nodes in the group. When the local load is greater than the threshold by some predefined percentage, which is set by the user, the last task in the run queue is transferred to the target node selected by the location policy.

Location Policy. This load balancing service will employ the minimum-loaded and random-target-host location policies. In the minimum-loaded policy, the target host selected for task transfer is the host with minimum load, which must also be less than the threshold value. In the random-target-host policy, when a task is to be transferred, a random host whose load is less than the threshold value is selected, and the task is transferred to that host. In both policies, the length of the run queue is used as the load index. Other application-specific load indices can also be used.

In both the policies explained above, the search for the appropriate target host can be restricted to nearby nodes or could span the entire the system. The former scheme is known as the nearest-neighbor load balancing scheme [17-19] while the latter, which in

general is hierarchical for scalability reasons, is known as the hierarchical load balancing scheme [18, 20-21].

6.1 Nearest-neighbor Load Balancing

In the nearest-neighbor load balancing scheme, the system is divided into groups and task transfers from an overloaded host to an underloaded host are limited to hosts within the group. Figure 6-1 illustrates the nearest-neighbor load balancing scheme. Here, group II does not have an underloaded node and hence no task is transferred from the overloaded node even though more than one underloaded node is available in group III. Thus, in the case of group III the excess load can be partitioned between the two underloaded nodes. Since tasks are transferred within the group, nearest-neighbor load balancing benefits from reduced communication overhead. The scheme may be better suited for grids, where resources are spread far away from each other and are in different security domains. Also, applications with large amounts of data to be transferred perform better with the nearest-neighbor load balancing scheme. However, depending upon the group size used in the system, there may be few if any neighbors to any given node. The scheme also fails to use the idle resources in adjacent groups.

This scheme can be effectively used in coordination with GEMS as the information service. The search domain for idle host can span the entire system when GEMS is configured as a flat service while the domain can be restricted to a selected group of nodes upon configuring GEMS as a layered service.

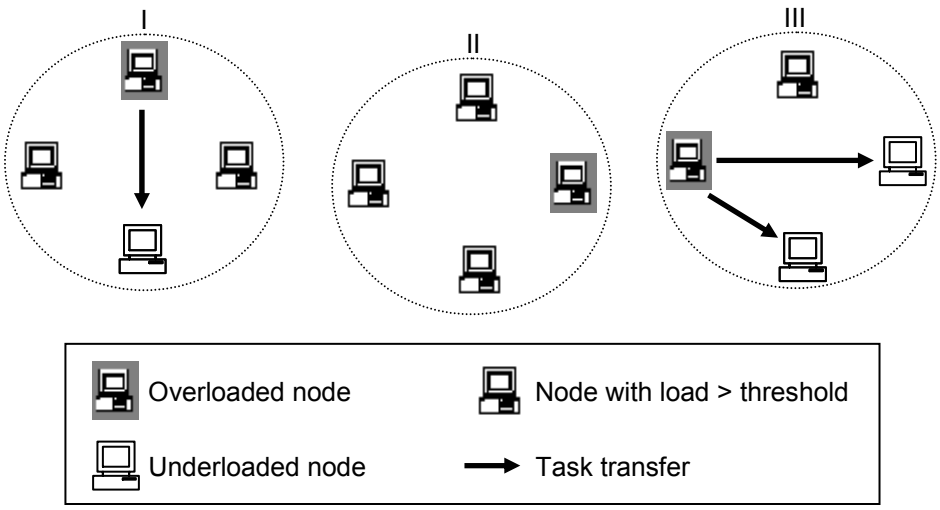


Figure 6-1. Example of the nearest-neighbor load balancing scheme

6.2 Hierarchical Load Balancing

Hierarchical load balancing is carried out in two stages. When a task is to be transferred, the load of the nodes within the group is compared with the aggregate load of the other groups in the system. In Figure 6-2, node 1 in group I is overloaded. The location policy compares the load of local nodes 2, 3, and 4 in group I with the aggregate load of groups II and III. Since the aggregate load of group II is less than the load of local nodes 2, 3, 4 and group III (aggregate load), the task is transferred to a random node in group II. The task is again reassigned to the minimum loaded node in group II. The communication overhead and latency are higher in this scheme, making it better suited only for long-running tasks. The task transfer involves all nodes in the system and leads to better balance and efficient resource utilization.

The hierarchical load balancing scheme can be used only when GEMS is configured as a layered service. The layered structure of the hierarchical load balancing scheme can be easily overlaid on the GEMS architecture with the groups of the load balancing scheme matching that of layered GEMS groups. Thus, using GEMS we have

developed a scalable, distributed and dynamic load balancing service with two popular dynamic load balancing schemes. The case study also illustrates that newer and more complex load balancing algorithms that use application specific load indices can be efficiently built on top of GEMS. The algorithms developed can use the simple API of GEMS to gather the load indices.

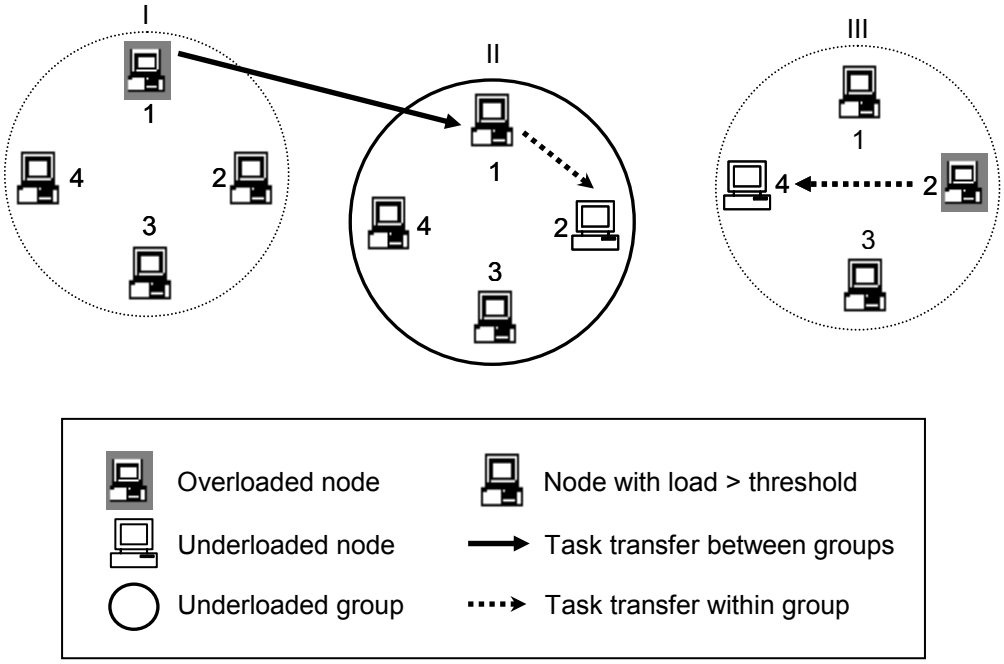


Figure 6-2. Example of the hierarchical load balancing scheme

CHAPTER 7
CONCLUSIONS

A scalable, distributed and dynamic resource monitoring service has been developed and the features of the service are elucidated in this thesis. The development of GEMS upon the gossip-style failure detection service presents various advantages. The gossip-based dissemination of the resource monitoring data provides fault tolerance and reduced overhead. Data consistency is also maintained, at no extra cost using the heartbeat mechanism of gossip-style failure detection service. The scalability is greatly improved by hierarchical design of the service, collecting data of individual nodes in the group to represent an aggregate view of the group.

The service is extensible, with provisions for adding new sensors and disseminating application-specific data. This feature can be used for developing new distributed applications that disseminate data through GEMS in a scalable and fault-tolerant manner. We also provide provisions for dynamic inclusion of new aggregation functions, helping applications in generating customized aggregate view of the group. Finally, the service is flexible offering both individual as well as aggregate view on the state of the system.

The major applications of GEMS are in the fields of system administration, scheduling and load balancing. GEMS effectively supports system administration through its extensibility and flexibility. To demonstrate the usefulness of GEMS for load balancing and scheduling middleware services, we developed a load balancing service as a case study using GEMS as the information service. The case study illustrates that new

middleware services that use GEMS for retrieving system and application specific data can be developed with considerable ease.

We also conducted a series of experiments to ascertain the scalability of the service in terms of its resource utilization. The experiments show that GEMS exhibits strong scalability versus resource utilization. We have shown analytically that the bandwidth consumption of the service is less than 60 Kbps for a system of 1024 nodes, and experimentally that CPU utilization is less than 2% for a system of 128 nodes. By carefully configuring the system group sizes to match that of the optimum group size, the resource utilization can be greatly minimized for a given system size.

Directions for future research include the addition of new performance measurement sensors and aggregation functions. The sensors currently use system calls and user commands to measure system parameters, which incur delay. The sensors can be redesigned to read system parameters directly from the kernel table for faster response. The resource utilization experiments were performed in this work using GEMS as a stand-alone service. Experimental studies at the application level can be performed by running GEMS in coordination with large distributed applications to ascertain the resource utilization measurements. A better and flexible user interface should be developed for representing monitored data in various user-friendly forms. The monitored information of GEMS combined with a graphical user interface can be used to develop a Single System Image (SSI) for heterogeneous clusters. Finally, GEMS currently supports only two layers of aggregation, which should be extended to support multiple layers and the dynamic insertion of new nodes into the system.

REFERENCES

1. K. Sistla, A. George, R. Todd, and R. Tilak, "Performance analysis of flat and layered gossip services for failure detection and consensus in heterogeneous cluster computing," Proceedings of Heterogeneous Computing Workshop (HCW) at the International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, April, 2001, Retrieved October 28, 2002 from the World Wide Web: http://www.hcs.ufl.edu/pubs/HCW2001.pdf.

2. K. Sistla, A. George, and R. Todd, "Experimental analysis of a gossip-based service for scalable, distributed failure detection and consensus," Cluster Computing, In press, Retrieved October 28, 2002 from the World Wide Web: http://www.hcs.ufl.edu/pubs/GOSSIP2001.pdf.

3. R. Wolski, "Dynamically forecasting network performance to support dynamic scheduling using the network weather service," Cluster Computing, January, 1998, Vol. 1, No. 1, pp: 119-131.

4. R. Wolski, N. Spring, and J. Hayes, "The network weather service: a distributed resource performance forecasting service for metacomputing," Journal of Future Generation Computing Systems, October, 1999, Vol. 15, No. 5-6, pp: 757-768.

5. International Business Machines Corporation, *IBM LoadLeveler: User's Guide*, International Business Machines Corporation, Kingston, New York, September, 1993.

6. Z. Liang, Y. Sun, and C. Wang, "Clusterprobe: an open, flexible and scalable cluster monitoring tool," Proceedings of 1st IEEE Computer Society International Workshop on Cluster Computing, Melbourne, Australia, December, 1999, pp: 261-268.

7. R. Buyya, "PARMON: a portable and scalable monitoring system for clusters," International Journal on Software: Practice & Experience, June, 2000, Vol. 30, No. 7, pp: 723-739.

8. R. Van Renesse, "Scalable and secure resource location," Proceedings of 33rd IEEE Annual Hawaii International Conference on System Sciences, Island of Maui, Hawaii, January, 2000, pp: 1209-1218.

9. R. Van Renesse, R. Minsky, and M. Hayden, "A gossip-style failure detection service," Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing Middleware, Lake District, England, September, 1998, pp: 55-70.

10. M. Burns, A. George, and B. Wallace, "Simulative performance analysis of gossip failure detection for scalable distributed systems," Cluster Computing, January, 1999, Vol. 2, No. 3, pp: 207-217.

11. J. Tan, "Cost-efficient load distribution using multicasting," Proceeding of 1st IEEE Computer Society International Workshop on Cluster Computing, Melbourne, Australia, December, 1999, pp: 202-208.

12. K. S. Chin, "Load sharing in distributed real time systems with state change broadcasts," IEEE Transactions on Computers, August, 1989, Vol. 38, No. 8, pp: 1124-1142.

13. S. Ranganathan, A. George, R. Todd, and M. Chidester, "Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters," Cluster Computing, July, 2001, Vol. 4, No. 3, pp: 197-209.

14. R. Subramaniyan, A. George, and M. Radlinski, "Toward a correct, complete and scalable gossip-style failure detection service for heterogeneous clusters," Technical report, High-performance Computing and Simulation Research Laboratory, University of Florida, Fall 2002.

15. H. C. Lin and C. S. Raghavendra, "A dynamic load balancing policy with a central job dispatcher (LBC)," IEEE Transactions on Software Engineering, February, 1992, Vol. 18, No. 2, pp: 148-158.

16. S. Zhou, "A trace-driven simulation study of dynamic load balancing," IEEE Transactions on Software Engineering, September, 1988, Vol. 14, No. 9, pp: 1327-1341.

17. M. Zaki, W. Li, and S. Parthasarathy, "Customized dynamic load balancing for a network of workstations," Journal of Parallel and Distributed Computing, June, 1997, Vol. 43, No. 2, pp: 156-162.

18. M. Willebeek-LeMair and A. Reeves, "Strategies for dynamic load balancing on highly parallel computers," IEEE Transactions on Parallel and Distributed Systems, September, 1993, Vol. 4, No. 9, pp: 979-993.

19. C. Xu, B. Monien, and R. Luling, "Nearest neighbor algorithms for load balancing in parallel computers," Concurrency: Practice and Experience, October, 1995, Vol. 7, No. 7, pp: 707-736.

20. I. Ahmed, “Semi-distributed load balancing for massively parallel multicomputer systems,” IEEE Transactions on Software Engineering, October, 1991, Vol. 17, No. 10, pp: 987-1004.

21. K. Antonis, J. Garofalakis, J. Mourtos, and P. Spirakis, “A hierarchical adaptive distributed algorithm for load balancing,” Technical report, University of Patras, 1998.

BIOGRAPHY

Pirabhu Raman was born in Madras, India. After completing his schooling at NSN higher secondary school he proceeded to do his bachelor's in electrical and electronics engineering at the P.S.G College of Technology, Coimbatore, India. On completion of his bachelor's, he moved on to the University of Florida to pursue his master's at the Electrical and Computer Engineering Department. He joined the High-performance Computation and Simulation (HCS) Research Laboratory in spring 2001 as a graduate research assistant, to perform research in high-performance computing. His research at the HCS Research Lab forms the basis for this work.