Cluster Computing 9, 101–120, 2006
© 2006 Springer Science + Business Media, Inc. Manufactured in The United States.

GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems

RAJAGOPAL SUBRAMANIYAN, PIRABHU RAMAN, ALAN D. GEORGE* and MATTHEW RADLINSKI
High-performance Computing and Simulation (HCS) Research Laboratory, Department of Electrical and Computer Engineering, University of Florida, P.O. Box 116200, Gainesville, FL 32611-6200

Abstract. Gossip protocols have proven to be effective means by which failures can be detected in large, distributed systems in an asynchronous manner without the limitations associated with reliable multicasting for group communications. In this paper, we discuss the development and features of a Gossip-Enabled Monitoring Service (GEMS), a highly responsive and scalable resource monitoring service, to monitor health and performance information in heterogeneous distributed systems. GEMS has many novel and essential features such as detection of network partitions and dynamic insertion of new nodes into the service. Easily extensible, GEMS also incorporates facilities for distributing arbitrary system and application-specific data. We present experiments and analytical projections demonstrating scalability, fast response times and low resource utilization requirements, making GEMS a potent solution for resource monitoring in distributed computing.

Keywords: grid, cluster, resource monitoring, fault-tolerance, gossip protocol, probabilistic dissemination

1. Introduction

Clusters built from commercial off-the-shelf (COTS) components exhibit a level of simplicity and cost effectiveness their conventional supercomputing brethren lack and, as such, have seen an increase in popularity in the recent past. Regardless, the heterogeneity and rapidly increasing size of such clusters exacerbates the chore of maintaining them. 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 and is more difficult to satisfy in a heterogeneous environment where resources vary in quantity and quality from one node to another. Harnessing such idle resources requires the knowledge of the health (e.g., liveness) as well as the performance (e.g., utilization) of the resources in consideration.

Failures plague clusters designed from stand-alone workstations and personal computers (i.e., COTS systems), and such systems require dedicated services to monitor nodes and report failures, allowing self-healing and check-pointing applications to restart dead processes. Distributed applications require a reliable, fast and scalable low-level health monitoring service. Speed of such processes is critical, as low detection times minimize the impact of failures on the system and enable quick recovery from faults with strategies such as checkpointing and process migration. However, minimizing failure detection time is a non-trivial issue, as a system-wide consensus on failures must be reached in a scalable fashion.

In addition to fault-free execution, cluster performance is determined by the utilization of resources such as the CPU, interconnect, memory utilization, etc. Performance monitoring provides accurate estimate of this resource utilization, which enables time-critical services and long running applications with process migration capabilities to distribute processes and tasks. In summary, resource-monitoring services serve as an information source for performance, health and available resource location and usage information. 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.

This paper presents a reliable and scalable gossip-enabled monitoring service called GEMS. The resource monitoring service with gossip-style communication addresses the challenges of clustering, failure detection and performance monitoring with low overhead. The service employs distributed consensus for fast and reliable failure detection with reaction times in the milliseconds range, even for larger systems with hundreds and thousands of nodes. Performance information is piggybacked on liveness information providing a distributed health and performance monitoring service. The GEMS Application Programming Interface (API) also allows users to share application data and introduce user-defined aggregators into the service for easy expansion. We experimentally analyze and mathematically model the service to provide performance projections for very large systems beyond the scope of our testbed. A node-insertion mechanism, which improves dynamic scalability, further enhances the service.

The rest of the paper is organized as follows. The next section discusses some relevant resource monitoring services. Section 3 describes the failure detection mechanism employed for health monitoring in GEMS. Resource performance monitoring as well as the dissemination and retrieval of monitored information are described in Section 4. Resource utilization experiments and performance results, demonstrating

*Corresponding author.
E-mail: george@hcs.ufl.edu

the scalability of the service, follow in Section 5. Finally, conclusions and directions for future research are presented in Section 6.

2. Related research

Several resource-monitoring services have been developed for heterogeneous clusters of workstations but few meet the scalability and extensibility requirements of such clusters. None of the services completely address the relevant issues such as fault-tolerance, network partitions, and addition of new nodes to the system, which are vital requirements for COTS-based clusters. These include services such as the Network Weather Service, Load Leveler, Cluster Probe, Parmon, Condor 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) [1,2], 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 (*InitForecaster, RequestForecasts*) 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 halts the storage and retrieval of data, to and from, the permanent medium. By contrast, data is distributed among all the processes of a group in the GEMS service and the monitored data is intact even if only one of the processes is alive.

ClusterProbe [3], 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. ClusterProbe is extensible and provides for multi-protocol communication, enabling clients with varied communication protocols to access monitored results. ClusterProbe employs a central monitoring server, which acts as the point of contact for the clients as well as the proxies. The server receives monitoring requests from the clients and generates the appropriate monitoring sessions for each agent. This tool also suffers from a single point of failure at the monitoring server and is not resilient against proxy crashes. ClusterProbe implements global event facility as an extension of the local event facility provided by Java to assist in failure detection. Failures are located by matching the state of the resources with the abnormal conditions. This method of failure detection is not robust and does not address transient failures as well as group failures.

Parmon [4] is another resource monitoring service developed at the Centre for Development of Advanced Computing (CDAC) in India. Parmon monitors an extensive array of system parameters using a 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 may not scale beyond hundreds of nodes due to the significant network overhead introduced by the monitoring requests along with response messages. No information is provided regarding the behavior of Parmon when monitoring requests are issued for failed resources. Hence the service may not be fault-tolerant and robust.

In Astrolabe [5], a scalable and secure resource location service developed at Cornell University, gossip-style communication is used for data aggregation similar to GEMS. The service uses simple timeouts for failure detection which is unreliable and increases the probability of erroneous failure detection. The service also does not address the issues related to network partitions. 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 data aggregation functions and the API are SQL-based queries, which have their advantages and disadvantages in terms of simplicity and scalability respectively. Finally, the authors provide an 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.

Load Leveler [6], a job scheduling service 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. The service may not be extensible or fault-tolerant.

Condor [7], a workload management system developed at the University of Wisconsin, is specialized for compute-intensive jobs. Condor provides a job queuing mechanism, scheduling policy and priority scheme. Though Condor is among the most popular job management services that are used presently, it is a batch system like Load Leveler and suffers from the same limitations such as single point of failure and bottleneck at the Condor master (central manager). However, it may be not be fair to compare the functionalities and design of Load Leveler and Condor with GEMS because both these two services concentrate more on job management and scheduling, and resource monitoring is not their primary emphasis. GEMS, on the other hand, has been specifically designed as a robust and reliable resource monitoring service that can provide lower-level services to tools such as Condor and Load Leveler.

In summary, a distributed monitoring service that periodically disseminates the monitored data is needed for faster response times. The service should be robust and susceptible to failures including network partitions and any other arbitrary failure. 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. Scalability constraints limit services based on traditional group communication mechanisms. Gossip-style services exploit the inherent advantages of gossip communication in terms of being very responsive, having no single point of failure, and being far more scalable and efficient than classical group communication methods.

Research involving random gossiping to promulgate liveness information [8–11] has demonstrated high-speed, low-overhead dissemination of system state information. Early on, gossip concepts were primarily used for consistency management of replicated databases, reliable broadcast and multicast operations. Van Renesse et al. first investigated gossiping for failure detection [8] at Cornell University. In their paper, they present three protocols: a basic protocol, a hierarchical protocol and a protocol that deals with arbitrary host failures and partitions. Researchers at the University of Florida extended the preliminary work at Cornell to build a full-fledged failure detection service. Burns et al. performed high-fidelity, CAD-based modeling and simulation of various gossip protocols to demonstrate the strengths and weaknesses of different approaches [9]. Ranganathan et al. introduced several efficient communication patterns and simulated their performance [10]. Also in [10], the authors proposed a distributed consensus algorithm, which formed the basis for the experimental analysis of various flat and hierarchical designs of gossip-based failure detection by Sistla et al. in [11,12]. The design described in [11,12] is an efficient conception of a gossip-style failure detection service and forms the basis of our work as well.

3. Resource health monitoring in GEMS

The basic idea behind gossip-style communication is synonymous with the word ‘gossip’. Personal information and opinions about others are not kept a secret and are spread around to others. The receivers of such gossip messages spread the information to few others whom they know. Likewise in computer systems and clusters, gossip-style protocols employ a similar type of information dissemination mechanisms. Nodes that are part of the system frequently exchange their perspective of the nodes in the system with other nodes.

Three key parameters involved in health monitoring, failure detection and consensus are the gossip time, the cleanup time, and the consensus time. Gossip time, or T_{gossip}, is the time interval between two consecutive gossip messages sent out by a node. Cleanup time, or T_{cleanup}, is the interval between the time liveness information was last received for a particular node and the time it is suspected to have failed. That is, if node 1 receives no fresh liveness information about node 2 in T_{cleanup} time, then node 1 will suspect node 2 to have failed. Finally, consensus time, or $T_{\text{consensus}}$, is the time interval after which consensus is reached about a failed node. The first two are input parameters configured for a particular GEMS-based failure detection system. The cleanup time is some multiple of the gossip time, and the time required for information to reach all other nodes sets a lower bound for T_{cleanup}. When gossip time and cleanup time are relatively small, the system responds more quickly to changes in node status. When they are relatively large, response is slower but resource utilization decreases as fewer messages are exchanged and processed. The third parameter, $T_{\text{consensus}}$, is a performance metric determining how quickly failures are detected and consensus is reached.

3.1. Flat gossiping

The important data structures maintained in each node of the system are the gossip list, suspect vector, suspect matrix and live list. The gossip list is a vector containing the number of T_{gossip} intervals since the last heartbeat received for each node. The suspect vector’s ith element is set to ‘1’ if node i is suspected to have failed, otherwise it is set to ‘0.’ The suspect vectors of all n nodes in the system together form a suspect matrix of size $n \times n$. Finally, the live list is a vector maintaining the health status of all the nodes in the system.

Every T_{gossip}, a node chooses another node in the system at random and transmits a gossip message to it via UDP sockets. Past research at the University of Florida [3,4] has shown that random gossiping is more scalable and efficient than other communication patterns such as round-robin and binary round-robin. A gossip message consists of the sender’s gossip list and suspect matrix and various headers. The suspect matrix sent by node $i$has the suspect vector of node i as the ith row. On receipt of a gossip message, the local suspect vector and suspect matrix are updated based on the heartbeat values provided by the gossip list. Low values in the gossip list imply recent communication.

Figure 1 illustrates how the data structures in a node are updated upon receipt of a gossip message from another node in the system. The T_{cleanup} for the 4-node system has been set to a value of 20 (i.e., 20 × T_{gossip}). Initially in the figure, node 0 suspects nodes 2 and 3 to have failed, as the heartbeat entries in gossip list corresponding to nodes 2 and 3 indicate values greater than T_{cleanup}, the suspicion time. The entries corresponding to nodes 2 and 3 in the suspect list reflect the suspicion. Likewise, node 1 also suspects node 3 to have failed as indicated by a corresponding entry in the suspect list. Node 1 does not suspect node 2, as only five gossip intervals, which is less than T_{cleanup}, have elapsed since the receipt of a message from node 2. The entries in the suspect matrix of node 1 also indicate that node 2 suspects the failure of node 3, which would have been indicated by a gossip message from node 2 to node 1.

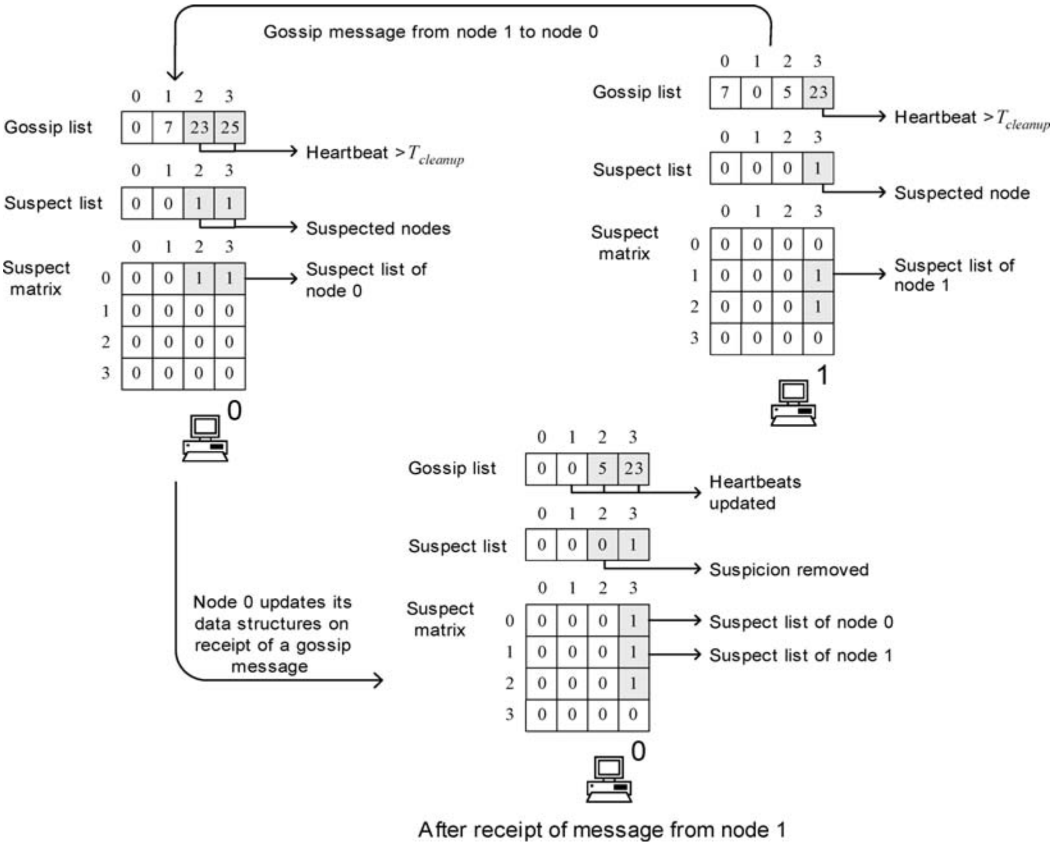


Figure 1. Illustration of data structure updates in a 4-node system with $T_{\text{cleanup}} = 20$.

On receipt of a gossip message from a node, say node Y, node 0 compares the heartbeat entries in the received gossip list with those in the local gossip list. A smaller value in the received gossip list corresponding to a node, say node X, implies more recent communication by node X with node Y. Node 0 subsequently replaces the entry in its gossip list corresponding to node X with the value in the received gossip list. A larger value in the received gossip list corresponding to node X implies that node X has communicated with node 0 more recently than with node Y. Then, the entry in the local gossip list is not modified. In figure 1, node 2 has communicated five gossip intervals ago with node 1, and 23 gossip intervals ago with node 0. On receipt of a message from node 1, node 0 updates the gossip list to reflect the fact that node 2 was alive as recent as five gossip intervals ago.

The suspect vector is next updated based on the updated gossip list. The suspicion entries are modified to reflect the present heartbeat values. In figure 1, the suspicion on node 2 is removed, while node 3 is still suspected. The suspect matrix is next updated, based on the modifications done to the gossip list. A new smaller heartbeat value corresponding to any node, say node Z, implies that the received message has node Z's more recent perspective of the system. In figure 1, heartbeat values of nodes 1, 2 and 3 have been changed in the gossip list, implying that node 1 has a more recent version of the perspective of other nodes (i.e., nodes 1, 2 and 3) in the system than does node 0. Subsequently, node 0 replaces the entries in the suspect matrix corresponding to nodes 1, 2 and 3 with those in the received suspect matrix. The entries corresponding to node 0 itself are also modified to reflect node 0's present vision of the system as given by its new suspect vector.

3.2. *Layered gossiping*

The layered design proposed here, a divide and conquer approach, divides the system into groups, which in turn are combined to form groups in different layers, with the number of layers dictated by performance requirements. The optimum choice for the number of layers to use in a system is described in Section 5.4. The notation 'L#' will be used hereafter to denote a layer with 'L' abbreviating layer and '#' layer number. Figure 2 illustrates the communication structure and system configuration of the multilayered design in a sample system having 27 nodes divided into three layers. The nodes are divided into groups, with three nodes in each L2 group. L2 groups are grouped themselves to form L3 groups. The system shown in the figure has three L2 groups in each L3 group. Groups of nodes in the higher layers are also treated similar to nodes in the first layer. Gossip lists, suspect vectors, and suspect matrices are maintained for each group in each layer similar to those for individual nodes. In a layered system, the gossip messages sent between nodes in the same group include the gossip list and suspect matrix of the nodes in the group, as in a flat system, along with the data structures

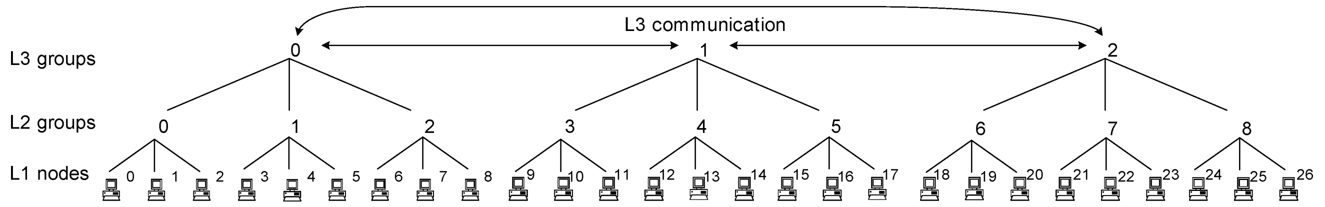


Figure 2. A sample multilayered system with 27 nodes and three layers.

involving the groups in all the upper layers. In general, a gossip message in layer k encompasses the information of the groups in all the layers above layer k. For example, in figure 2, a gossip message between node 0 and node 1 will include information corresponding to the nodes 0, 1 and 2 along with information corresponding to L2 groups 0, 1, 2 and L3 groups 0, 1 and 2. In order to communicate group information to other groups, nodes in each group loosely take turns by applying modulo arithmetic to the iteration counter maintained by GEMS. The iteration counter maintains the number of T_{gossip} intervals elapsed since the service has started in a node. When the remainder that is obtained on division of the iteration counter value by the number of nodes in the group is equal to the node number (within the group), the node communicates with other groups. This scheme maintains the distributed nature of the service without the necessity to choose group leaders and hence avoids single points of failure. An alternative is to simply apply a random-number generator at each node in the group so that each node has an equal probability of communication with another group, and on average one node does so every gossip interval. Consensus about the failure of a node is restricted to the L2 group in which the failure occurred and is propagated to the rest of the system by both broadcast and live list propagation, as described in Section 3.3.1.

3.3. Consensus on health failures

Primitive failure detection services work on basic timeout mechanisms. However, such services are vulnerable to network failure, delays and message losses. Gossip-style failure detection, though fully distributed, is not immune to false failure detections especially with the random pattern of gossip messages. In order to obtain a consistent system view and prevent false failure detections, it is necessary for all the nodes in the system to come to a consensus on the status of a failed node.

3.3.1. Design of consensus algorithm

Whenever a node suspects that any other node in the system may have failed, it checks the corresponding column of its suspect matrix to consult the opinions of other nodes about the suspected node. If more than half the number of presently unsuspected (live) nodes, i.e., a majority of the live nodes, suspect a node then the node is not included in the consensus. The opinion of the masked node is discarded. The majority check prevents false detections from affecting the correctness of the algorithm by ensuring that only faulty nodes are masked. Should all the other nodes agree with the suspicion, the suspected node is declared failed and the information is broadcasted to all the other nodes in the flat system. In the case of layered gossiping, the consensus is localized within a group and the failure is broadcasted to all the nodes in the system. Figure 3 illustrates the working of the consensus algorithm in a 4-node system with one failed node, node 0. Node 2 sends a suspect matrix to node 1 indicating that it and node 3 suspect node 0 may have failed based in part upon earlier messages received from node 3. Node 1, which already suspects node 0, updates its suspect matrix on receipt of a gossip message from node 2 and finds that every other node in the system suspects that node 0 has failed. The suspect matrix is updated as explained in the previous section. Thus, a consensus has been reached on the failure of node 0 and this information is broadcasted throughout the system. All the nodes subsequently update their live lists to indicate the status of node 0.

Since UDP broadcasts can be unreliable, an alternate method of intimating the failure of a node is necessary. One solution is to propagate the live list of a group by appending it to inter-group gossip messages. Every node in the system maintains the live list of the nodes within its own group, that is, a local live list. The local live list is generally consistent within a group, thanks to the consensus algorithm. The local live list is propagated to other groups and is also consistent irrespective of which node in the group sends the gossip message. On receipt of a gossip message from a different group, a node identifies the sender's group and updates the live list corresponding to the group. So the *L2 messages* will now include the live list of a group along with the gossip list and suspect matrix of the groups. In the multilayered design, the first-layer live list is appended to all higher-layer gossip messages.

Propagating a failure message through transmission of the live list as described above takes much more time than a broadcast. However, live list transmission is done *in addition to* a broadcast, so only the nodes that missed the broadcast need to be updated. Thus, the speed with which the nodes are updated with this additional method is not an important factor here. Being an input parameter, the frequency with which second and higher layer gossip messages are transmitted may be set up to be the same as the frequency with which first layer gossip messages are transmitted, thus propagating the live list more frequently, with little increase in overhead. Ultimately, every node will know the status of every other node in the system, regardless of the reliability of UDP broadcasts, and

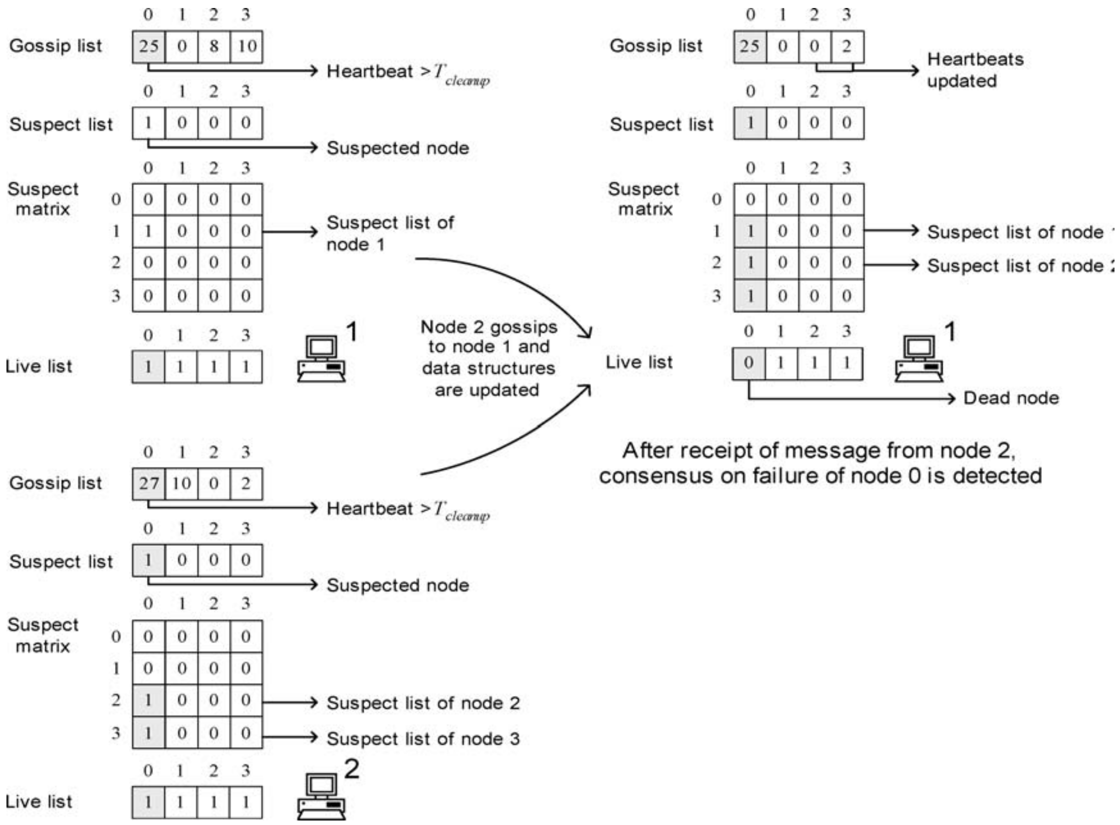


Figure 3. Illustration of consensus algorithm on a 4-node system.

the system will exhibit strong completeness. Presently, we are developing new and fast failure intimation mechanisms that can work in grid-like systems where broadcast is not possible.

3.3.2. Network partitions

Byzantine failures like network partitions, link failures and simultaneous failures challenge the correctness of the service. The majority requirement to mask faulty nodes from the consensus is reasonable only in cases when no bizarre failures occur and the system is running smoothly, except for a few minimal failures.

When a failure divides a system into two or more partitions, the partition with more than half the number of nodes in the entire system can always use the consensus algorithm successfully to detect the failure of the nodes in the other partitions. However, the nodes in smaller partitions cannot detect the failure of nodes in larger groups due to a lack of the majority required for consensus.

Timeout, the basis behind the idea of suspicion and failure detection, can be used to modify the consensus algorithm to overcome such Byzantine failures. Along with the normal procedure for detecting the failure of nodes, a user-definable timeout is used on the duration for which a node is suspected. For any node j suspected, the jth column of the suspect matrix is repeatedly checked at periodic intervals to verify if the entries are updated. A change in any entry $[i,j]$ of the suspect matrix indicates communication with node j indirectly via node i. By contrast, if the node is suspected even after the timeout period and none of the entries in the jth column have been updated, a communication failure has likely occurred. Subsequently, the live list is updated to reflect the failure of communication with node j. Whenever there is a failure of any network link and henceforth a partition in the network, a few nodes might end up in the smaller partitions. Such nodes in the smaller partition use the enhanced consensus algorithm to identify the other nodes in their partition without being affected by the requirement for a majority check.

Figure 4 illustrates the consensus algorithm modified to use timeouts. The figure depicts a system with five nodes, partitioned into two groups of three and two nodes. The group with three nodes (i.e., nodes 2, 3 and 4) will use the consensus algorithm as usual to detect the failure of the other two nodes (i.e., nodes 0 and 1). In the other group, the majority requirement will not be satisfied and normal consensus will not work with just two nodes. However, the suspect matrix in node 1 will indicate the suspicion of the other three nodes, while the update check indicates no update. Node 0 will also indicate through its suspect vector that it has not received any updates for the three nodes in question. Thus node 0 and node 1 detect the network partition and consider the other nodes as failed. The same timeout mechanism is used by node 0, which detects that node 1 has broken communication with the other nodes in the system. Nodes 0 and 1 can now form a new logical system with size two, if their services are still required.

When many links fail simultaneously creating multiple partitions in the network, the consensus algorithm with timeout

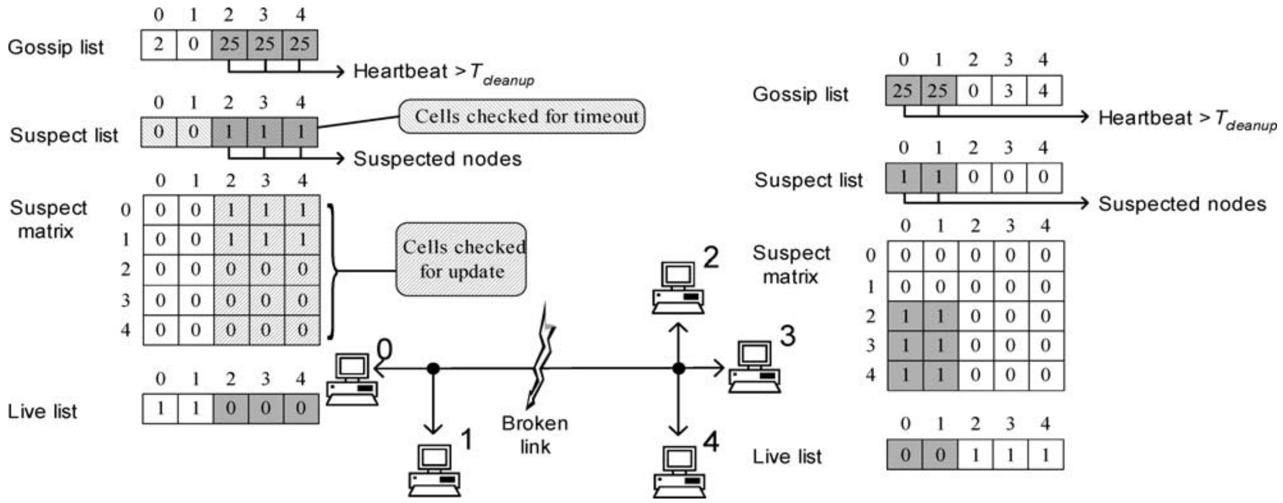


Figure 4. A flat 5-node system, with a broken link and using timeout-based consensus algorithm.

is used by all the nodes in the partitions that do not have the required number of nodes to use regular consensus. When the system is divided into two or more partitions of equal size, the timeout mechanism will identify the partition for all the nodes. The timeout-based algorithm is used to detect the partition of groups as well if a hierarchical failure detection service is employed. When a failure occurs, regardless of the nature of the failure, the modified algorithm helps the application using the service degrade gracefully or terminate, saving system resources. The deadlock resulting from a partition is avoided, and system-wide consistency is maintained.

3.3.3. Group failures

Failures of groups could be due to a partition in the network, splitting the network into two or more components, or due to the individual failures of all nodes in a group. Bulk failures go undetected if group failures are not addressed, calling into question the correctness and completeness of the service.

The solution is to detect such failures with a group-level consensus in the layer corresponding to the failure. On a group failure, the other groups with the same parent group in the next higher layer as the failed group use the standard consensus algorithm to detect the failure. All the subgroups and nodes within the failed group are also declared failed. However, it should be understood that a group is considered dead only when all the nodes in the group die. A group with even one live node is considered alive, since from the application's point of view, that one live node can share resources with the other groups. As an example of consensus on a group failure, as shown in figure 2, detection of failure of group 2 in L3 requires the consensus of L3 groups 0 and 1, which involves consensus of L2 groups 0-5, which requires the agreement of nodes 0 through 17.

4. Resource performance monitoring in GEMS

Gossip-style health monitoring is extended in GEMS to monitor the performance of various resources in the system. The performance information is piggybacked on health information, using the health monitoring service as a carrier to reduce overhead. The gossip heartbeat mechanism, which is used as part of health monitoring, is used here to maintain data consistency. In this section, we discuss the basic architecture for performance monitoring, its components, and the mechanism by which our service guarantees data consistency.

The performance monitoring module is made up of two main components: the Resource Monitoring Agent (RMA) and the Gossip Agent (GA). The GA basically refers to the resource health monitoring and failure detection part of GEMS discussed in the previous section. The RMA and GA should be active on each node that forms part of the resource monitoring service. The RMA gathers performance data monitored from the system and forms a Monitor Data Packet (MDP), which is forwarded to the GA. The GA piggybacks the MDP onto the 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. Figure 5 shows the exchange of MDPs in the resource monitoring service.

The RMA forms the basic block of the service and is composed of the sensors, the communication interface and the Application Programming Interface (API). Figure 6 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.

4.1. Sensors

The sensors interact with the hardware and applications to gather resource data, which forms the Management Information Base (MIB). GEMS has two types of sensors: built-in

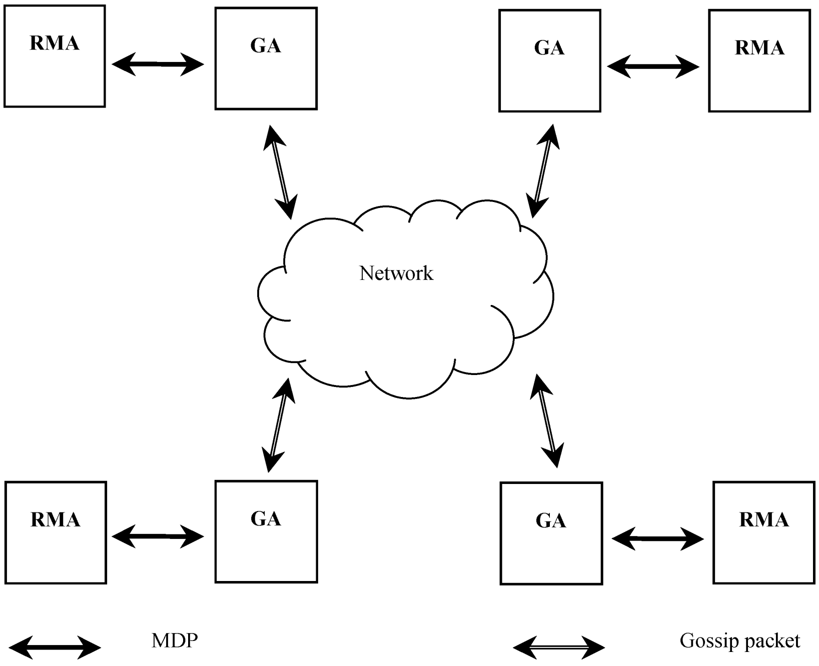


Figure 5. Illustration of MDP exchange in GEMS.

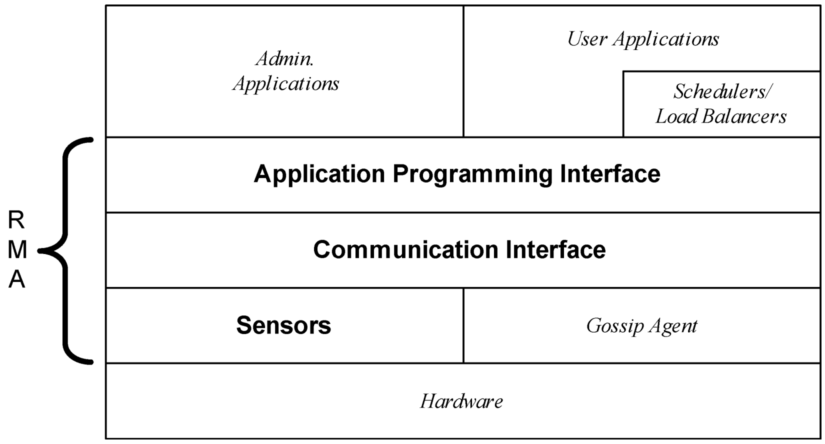


Figure 6. Components of the RMA.

sensors and user-defined sensors. Built-in sensors, which are provided with the service, actively measure the following performance parameters through the operating system services and system calls:

- *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 7 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

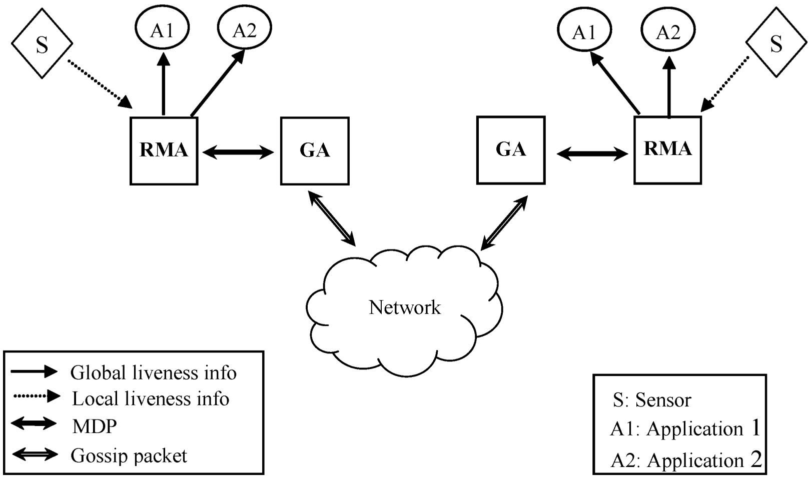


Figure 7. Example scenario describing the user data ID.

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 7, 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 RMA, 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 MDP, updating the MIB with newly received data and executing the aggregation functions to generate the aggregate MIB (AMIB).

Aggregation functions use the monitored data of all nodes in the group, to generate a single aggregate MIB representing the whole group. The AMIB is generated and maintained independently by every node in the group. Figure 8 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

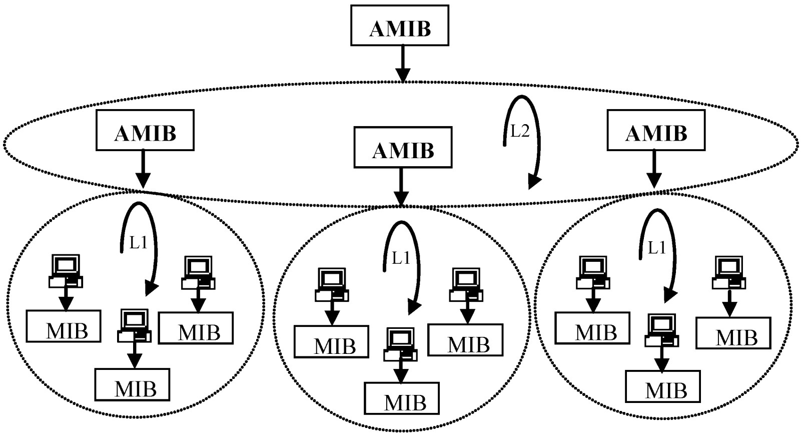
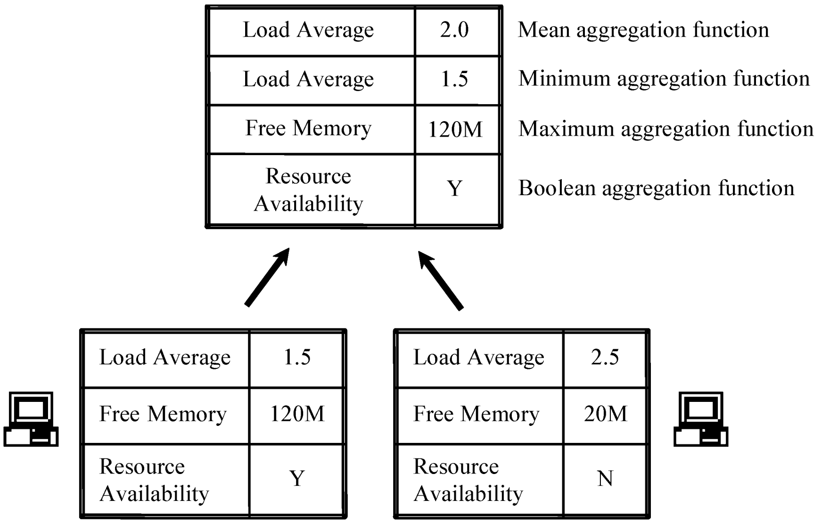


Figure 8. Structure of performance monitoring in GEMS.



Load Average	2.0	Mean aggregation function
Load Average	1.5	Minimum aggregation function
Free Memory	120M	Maximum aggregation function
Resource Availability	Y	Boolean aggregation function

Load Average	1.5
Free Memory	120M
Resource Availability	Y

Load Average	2.5
Free Memory	20M
Resource Availability	N

Figure 9. Illustration of the built-in aggregation functions.

MIB is exchanged within the group through L1 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 L2.

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 functions are provided with the service. The functions are identified by specific IDs similar to user data. Figure 9 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

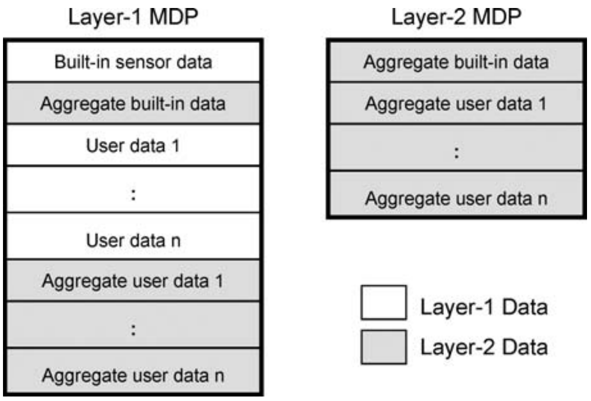
The aggregation functions provided with the service are limited and applications might require new modes of aggregation that are better suited for their data. Features have been added for introducing new user-defined aggregation functions into the system. Similar to the built-in aggregation functions, IDs are used for identification of user-defined aggregation functions. These aggregation functions are dynamically loaded into the service without the need for recompilation or restart of the service.

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 10 shows the layout of L1 and L2 MDPs in a two-layered system wherein the L1 packet contains the aggregate data of only the L2 groups. However, in a multi-layered system with more than two layers, the L1 packet will have the aggregate data of the groups in all the layers above L1. Thus, size of the L1 packet depends on the number of nodes/groups in each layer while that of the layer-*n* packet depends only on the number of groups in layer *n* and higher. L1 packets, therefore, are the largest packets in the system and are exchanged only within the local groups, saving network resources.

4.2.4. Data consistency

MDPs are updated using the gossip heartbeat values similar to the gossip list update. Whenever a new MDP is received, the



Layer-1 MDP
Built-in sensor data
Aggregate built-in data
User data 1
:
User data n
Aggregate user data 1
:
Aggregate user data n

Layer-2 MDP
Aggregate built-in data
Aggregate user data 1
:
Aggregate user data n

□	Layer-1 Data
▨	Layer-2 Data

Figure 10. Structure of the MDP.

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 11. Example illustrating update of monitor data packets.

communication interface merges the received contents with data previously present in the local RMA. Figure 11 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 API consists of a set of easy-to-use dynamic library functions. The API forms the principal mode of communication between the application and the RMA. The functions are broadly categorized into initialization, control and update functions.

4.3.1. *Initialization functions*

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

4.3.2. *Control functions*

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

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 3 shows the various monitor update functions, their operation and their return values.

Table 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

Table 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

4.4. *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

Table 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

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. Listed below is the pseudo-code detailing the procedure for the dissemination of user data with appropriate API functions.

```
userdata_id = gems_userdata_init( );              //get IDs from RMA — only one
aggfn_id = gems_aggfn_init( );                    //of the nodes should do this
while (1)
{
    data = sensor ( );                            //update my local data
    gems_senduserdata (userdata_id, aggfn_id, data); //send my local data
    gems_recvuserdata (userdata_ id);             //receive data of all hosts from RMA
    sleep (1);
    if(condition)
      gems_userdata_stop (userdata_id);           //stop dissemination
  }
```

4.5. Dynamic system reconfiguration

The scalability and efficiency of any service might depend on the reconfiguration facilities that are part of the service. Likewise, the dynamic scalability of the gossip service depends on the ability to join a new node into the system on the fly. Without dynamic reconfiguration facilities, the service would be unable to support applications which may require or benefit from an incremental allocation of resources.

Dynamic scalability of a service, the ability to expand the size of the system without a restart of the application is largely dependent on the abilities and performance of the reconfiguration software embedded in the service. This requires the expansion of the GEMS service as well to support the addition of new nodes into the system dynamically without any hindrance to regular services. Support for node-insertion also enables the service to be easily used with load balancing and scheduling services, wherein new nodes frequently join and leave the system. The remainder of this section discusses the mechanism to insert nodes into the system during execution.

4.5.1. Node re-initialization

Node insertion can be coarsely defined as *the addition of a new node into the cluster or inclusion of a node already part of a cluster into another group of nodes running a specific application*. The node insertion mechanism is used to add a new node into the system, or to re-initialize a node which previously failed and has since recovered. Re-initialization is faster compared to inserting a new node, as no data structures need to be rebuilt.

To re-initialize a node, the gossip service must be restarted on the node after it has recovered. The node starts to communicate gossip messages with other nodes as usual. The node which receives the first gossip message from the restarted node is called the *broker* node. The broker node realizes from its live list that a failed node has come back to life again, and it takes the responsibility of broadcasting the information to all the other nodes in the system. On receipt of the broadcast, each node's gossip list is modified to indicate recent communication. Their suspect vectors and suspect matrices are modified to remove suspicion, and the live lists are changed to show the live status of the node. The broadcast avoids inconsistency in the system when two or more dead nodes come back alive simultaneously. The restarted node determines the status of all other nodes through normal gossiping.

4.5.2. Design of the node-insertion mechanism

Many general node-insertion and reconfiguration mechanisms relevant to gossip-style service designed in the past were considered to provide deadlock-free and consistent node-insertion software. Approaches used in mobile ad-hoc networks were also studied as they involve very frequent reconfiguration. Realizing an insertion time as short as possible and leaving the system uninterrupted is critical. We have developed a fully distributed approach with low insertion times. The rest of this section discusses the design of the node-insertion mechanism built for GEMS.

The joining algorithm involves six different phases. During phase 1, the joiner node tries to identify a sponsor node. The new node can choose any node in the group in which it wants to join as its sponsor. Multiple requests to a sponsor may be done sequentially, should the first attempt fail due to timeout or a negative response. Steps 1 and 2 in figure 12 form the first phase. During the second phase, the joiner requests the sponsor to allow it to join the sponsor's group and goes into a wait state. During the third phase, the sponsor acquires a global lock to ensure that only a single join is in progress at any instant

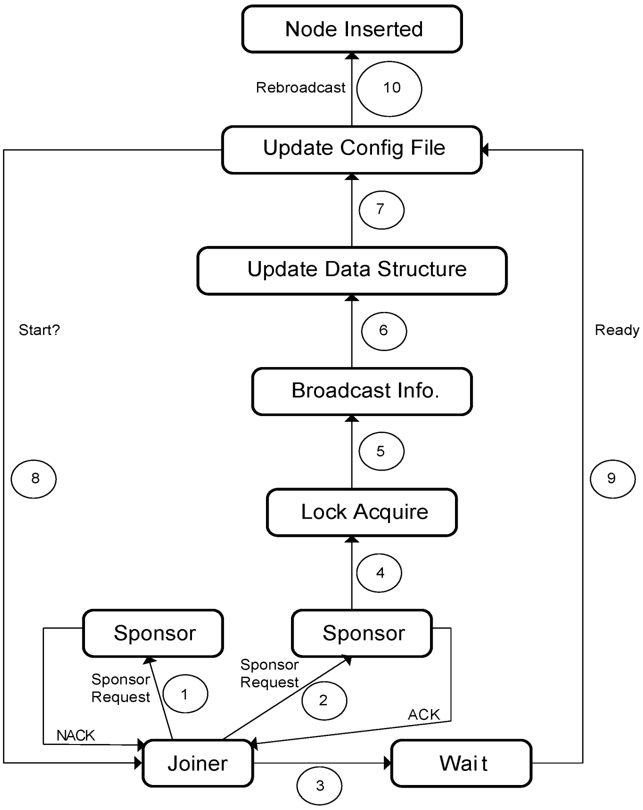


Figure 12. Sequence of steps illustrating the join of a new node.

of time. The locking mechanism is inspired by the global-update (GUP) lock of the Microsoft Cluster Service (MSCS) [13]. One node in the system plays the role of a lock manager, with its identity known system wide. The sponsor node queries the lock manager for the lock, and proceeds to insert the new node after obtaining the lock. The lock manager reclaims the lock after the insertion of the new node. When multiple nodes attempt to join the system simultaneously, the lock requests are maintained in a queue at the lock manager, and the lock is released to the sponsors in the order in which their requests were received. Should the lock manager fail, the next node in the system takes up the role, providing fault tolerance. There could be speculations that the hot spot at the lock manager will limit the scalability of the mechanism. Considering the fact that node insertions are infrequent, and the chances of simultaneous insertions of large number of nodes which create the hot spot are very low, the locking mechanism is scalable. However, the locking mechanism suffers from one limitation in that the sponsor nodes that are waiting to acquire the lock have to resubmit their lock request to the new lock manager when the present lock manager fails.

During the fourth phase, the sponsor broadcasts the identity of the new node to all the nodes in the system. The various data structures are modified in all the nodes to include the recent addition. The sponsor also takes the responsibility of modifying the shared configuration file to reflect the inclusion of the new node. Steps 5, 6 and 7 in the figure form the fourth phase. During the fifth phase, the sponsor sends current system information to the joiner and waits for the joiner's acknowledgement. On receipt of an acknowledgement, the sponsor broadcasts restart instructions to all nodes including the new node and releases the lock in the sixth phase. Whenever a sponsor node fails during the process of insertion, another node in the system takes up the responsibility of sponsoring the join. This step avoids deadlocks during the join. The choice of the new sponsor is made based on the order of nodes in the configuration file, which avoids unnecessary contention and race conditions. The design has been tested to be robust under all critical conditions.

It is worth noting here the reasons why we require a special mechanism to insert nodes into the system rather than letting nodes join asynchronously by sending gossip messages. The GEMS service is designed with fixed array data structures with the order of the nodes represented in the arrays remaining the same in each node throughout the system. The service relies on a common configuration file shared by all the nodes for this order. The advantage of this approach is that the size of the gossip messages is kept small as the messages merely contain the information (opinions) about the nodes in a specific order rather than explicitly storing and equating information belonging to each node. We are presently developing techniques to asynchronously join nodes in the service while keeping the size of the gossip messages small.

5. Experiments and results

We conducted a series of experiments involving the measurement of failure detection times, network and CPU utilization, and node insertion times to study the failure detection speed, scalability and performance of the service. The experiments were conducted on a PC cluster of 150 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.

5.1. Failure detection time experiments

Figure 13 illustrates the dependence of failure detection and consensus time on the size of a group in a two-layered system. The figure validates the results provided in [12], as well as extends the experiments for larger systems sizes. For a given system, the best consensus time is achieved by setting the T_{cleanup}parameter to the lowest possible value, called optimal cleanup time [12], below which true consensus cannot be reached. Selecting a value below this minimum for T_{cleanup} will increase false failure detections and make consensus impossible. For a fixed group size, consensus time is entirely independent of system size. Large group sizes result in long consensus times, while small group sizes yield lower consensus times. The smaller the group size, the faster the failure

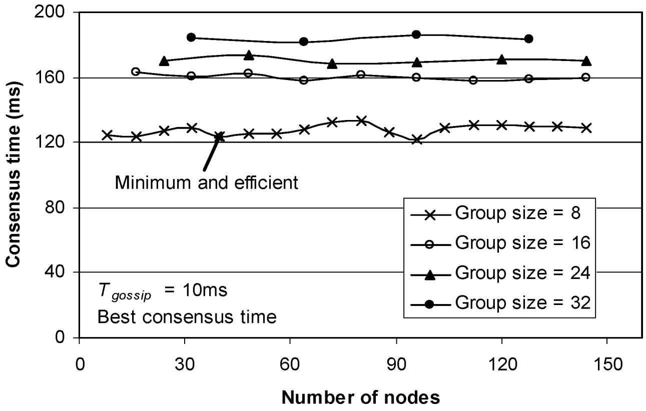


Figure 13. Detection time of node failures in a two-layered system.

detection, since fewer nodes are required for consensus. However, the size of L2 gossip messages increases with the number of groups (i.e. smaller group size implies more groups for a fixed system size), so a tradeoff exists between overhead and failure detection time when dealing with two-level gossiping. Rather than increasing the number of groups, the number of layers may be increased, thereby mitigating this effect. Using a multilayered design, the group size can be kept small without overhead constraints by increasing the number of layers. Thus with a multilayered setup, GEMS can detect node failures in as little as 130 ms irrespective of system size while most of the other existing services described in Section 2 take seconds to detect failure of a node.

Failure detection of groups is similar to that of nodes, except that subgroups participate in consensus instead of individual nodes. Figure 14 compares the detection time of a L2 group in two-layered and three-layered systems of the same size. Increasing the number of layers can keep detection time for group failures low. With the L2 group size fixed at four, the number of L2 groups must increase with system size. This increase in the number of L2 groups increases the detection time of a group failure in a two-layered system, as more groups participate in consensus. In the case of a three-layered system with a L3 group size of three, every three L2 groups form a L3 group. When a L2 group fails, only two other groups must come to a consensus, speeding up the process.

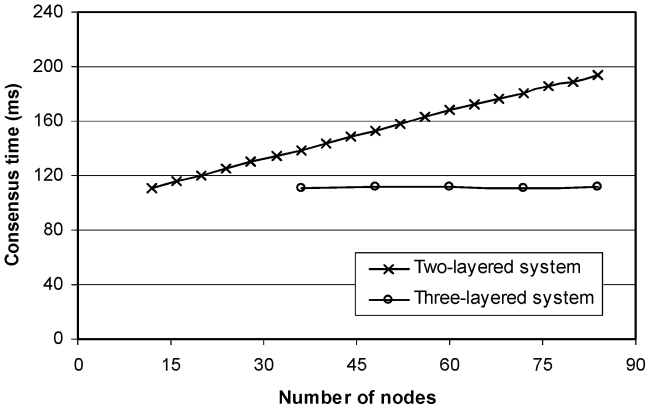


Figure 14. Comparison of detection time of group failures in layered systems.

5.2. Network utilization experiments

What are the performance costs of using this service, and is this service scalable? We measured the required network bandwidth using a packet-capturing tool called *Ethereal* available under the GNU public license. The network utilization measurements discussed here are only for the basic resource health monitoring module unless otherwise specified. Since the performance data that is piggybacked on health data is variable, the size of the data representing the monitored performance information is also not a constant and hence it would not be appropriate to include them in the general network utilization measurements.

Figure 15(a) shows the variation of bandwidth requirement per node for various group sizes. With a fixed L2 group size, an increase in system size increases the number of groups, increasing the L2 component of network utilization while the L1 component remains constant. The L1 network utilization component dominates the L2 component for small systems, while network use due to L2 traffic stands out for larger systems. With larger group sizes, the L2 component remains small, even for larger systems. Note the sharp increase in required bandwidth whenever the number of groups (system size ÷ group size) crosses a multiple of eight. Steep increases can be seen when system size moves from 64 to 72 and 128 to 136 for a group size of eight, and 128 to 144 for a group size of 16. Since gossip data is a sequence of bits, and packets are transmitted as bytes, an entire additional byte is required whenever the number of bits required crosses a multiple of eight.

Figure 15(b) illustrates the same behavior in a three-layered system for various L2 group sizes. When the number of L3 groups is fixed at two, each L3 group contains half the L2 groups. For example, with group size eight, a system with 128 nodes will have 16 L2 groups and two L3 groups, each of size eight. In a three-layered system, the L2 component also remains constant for a fixed L3 group size. When the system size increases from 128 to 144, while the L2 group size remains fixed at eight, the L3 group size increases from eight to nine with a steep rise in required bandwidth.

Results in figure 16 demonstrate the improved scalability of the three-layered system over the two-layered system, justifying the development of a layered service supporting any number of layers. In figure 16(a), the bandwidth requirement per node is compared for two- and three-layered systems with L2 group size fixed at eight.

For smaller systems ($<64 = 8 \times 8$, here), additional layers increase bandwidth due to necessary but extraneous communication in the higher layers. For larger systems (>64 here), bandwidth required per node in a three-layered system is less than that in a two-layered system, since fewer L2 groups communicate frequently. Reducing the size of a L3 group further

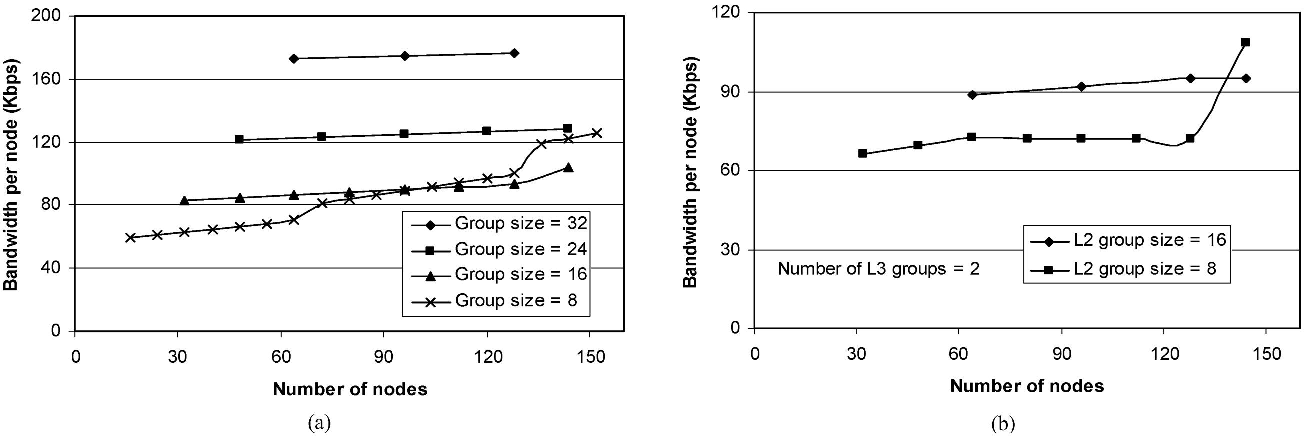


Figure 15. Variation of bandwidth requirement per node with group size in (a) two-layered system and (b) three-layered system.

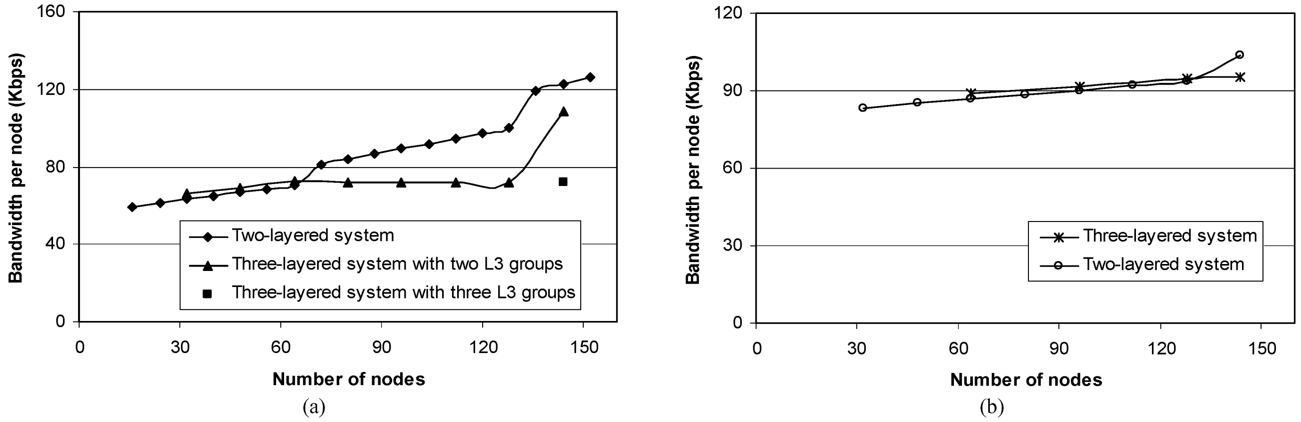


Figure 16. Comparison of bandwidth requirement per node in two and three-layered systems with (a) L2 group size = 8 (b) L2 group size = 16.

reduces the bandwidth, resulting in an even greater improvement over a two-layered system. Figure 16(b) illustrates the situation when the L2 group size fixed at 16. Here, the three-layered system performs better than the two-layered system for system sizes greater than 128, when the number of L2 groups increases beyond eight.

5.3. Processor utilization experiments

Processor utilization of the service is computed by measuring the number of CPU cycles consumed per T_{gossip}. The same 150-node PC cluster that was used for failure detection time and network utilization experiments was used for processor utilization experiments as well. The number of nodes used for the experiments was varied from 8 to 144. Processor utilization was measured in several nodes, each equipped with a 733 MHz Intel Pentium-III and 256 MB of memory.

Results in figure 17 demonstrate the scalability of three-layered systems over two-layered systems for larger system sizes. From the figure, it is seen that CPU utilization results follow the same trend as network utilization results with steep increases when the number of nodes or groups in a layer (that form a group in the next higher layer) crosses a multiple of eight. For example, in a two-layered system with L2 group size of eight, steep increases can be seen when system size increases from 64 to 72 and 128 to 136 because the number of L2 groups increases from 8 to 9 and 16 to 17 respectively. Figure 17(a) compares the CPU utilization of two- and three-layered systems with L2 group sizes fixed at eight. For smaller systems ($<64 = 8\times 8$, here), additional layers increase the CPU utilization due to necessary but extraneous computations for the higher layers. For larger systems (>64 here), CPU utilization per node in a three-layered system is less than that in a two-layered system, since computation involves fewer L2 groups. For a 144-node system, three-layered gossiping requires only about half of the CPU utilization required by two-layered gossiping with a L2 group size of eight. Similar to the network utilization case, reducing the size of a L3 group further reduces the CPU utilization, resulting in an even greater improvement over a two-layered system. Figure 17(b) illustrates the same situation when the L2 group size is fixed at 16. The crossover can be seen when system size crosses 128 and the number of L2 groups increases from eight to nine. In general, as L2 group size increases, the crossover shifts to the right (i.e., the curves cross at a larger system size). In any case, the curves diverge from each other after the crossover.

Here again, the utilization measurements do not involve the resource performance data of GEMS for the same reasons mentioned in the network utilization experiments. However, though not presented here, it was experimentally found that

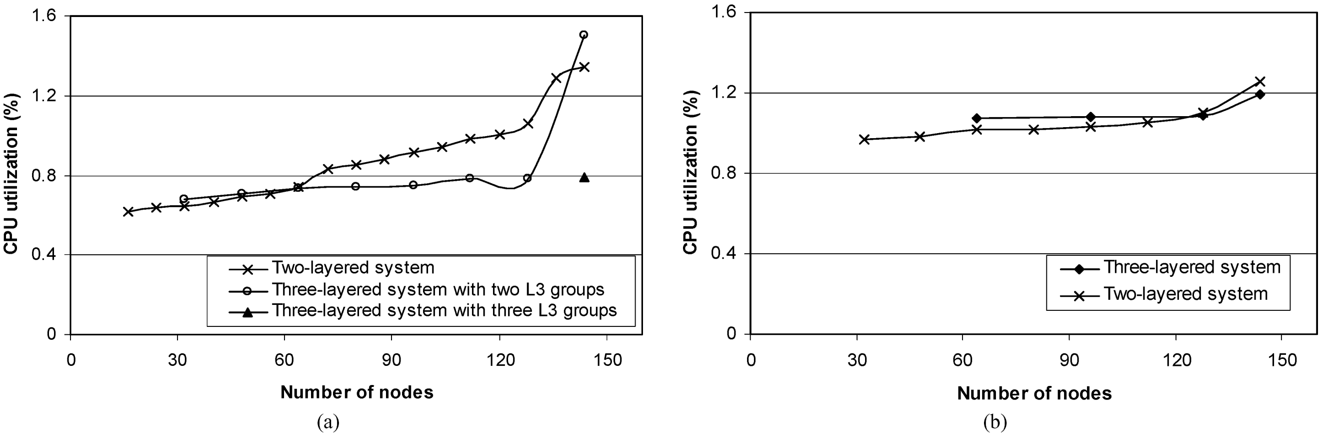


Figure 17. Comparison of CPU utilization in two and three-layered systems with (a) L2 group size = 8 and (b) L2 group size = 16.

the CPU utilization is not more than 2% when the module monitoring the performance of basic system parameters is included.

It can be seen from the utilization results in figure 17 that GEMS uses less than 2% of the CPU. It should be noted that when machines with faster processors are used, the processor utilization will be even lower since the execution time is lower in faster machines for the same workload. Hence, GEMS is shown to be highly efficient in terms of processor utilization.

5.4. Timing results and projections for node insertion

During the insertion of a node, gossip communication is stopped between nodes to avoid inconsistency, when a few nodes in the system will use the newly modified configuration file while others use the older unmodified version of the configuration file. If a failure occurs during this time, it will be detected and reported throughout the system only after communication resumes. Thus, minimizing this communication stoppage time, which depends on node-insertion time, is critical to maintain faster failure detection. The time to insert a new node was measured under various setups to demonstrate the efficiency of the mechanism designed.

Figure 18 gives the time to insert a new node in a system with two layers for various group sizes. The time to insert a node is the time from when the sponsor receives the request to insert the node until the moment when the communication is restarted after the node is inserted. The insertion time is simply the time taken at the sponsor node to execute the function for inserting a node. Due to the limitations in the size of the testbed, the insertion time is measured by simulating the insertion function to study the efficiency of the mechanism in large systems. The data structures are initialized to provide the impact of any required system size. The time taken to execute the function is measured by changing the system size and group size. The same nodes that were used for processor utilization measurements were used here as well. Each point in the plot represents an average of 10 different trials.

Insertion times are in the microseconds even for very large systems as compared to other similar schemes addressed earlier, where insertion times are on the order of tens of seconds. We observe that insertion times decrease with an increase in group size, because the number of groups decreases as the group size increases. Figure 18(b) illustrates that the pattern followed by the insertion time is the same irrespective of the group in which the node is inserted.

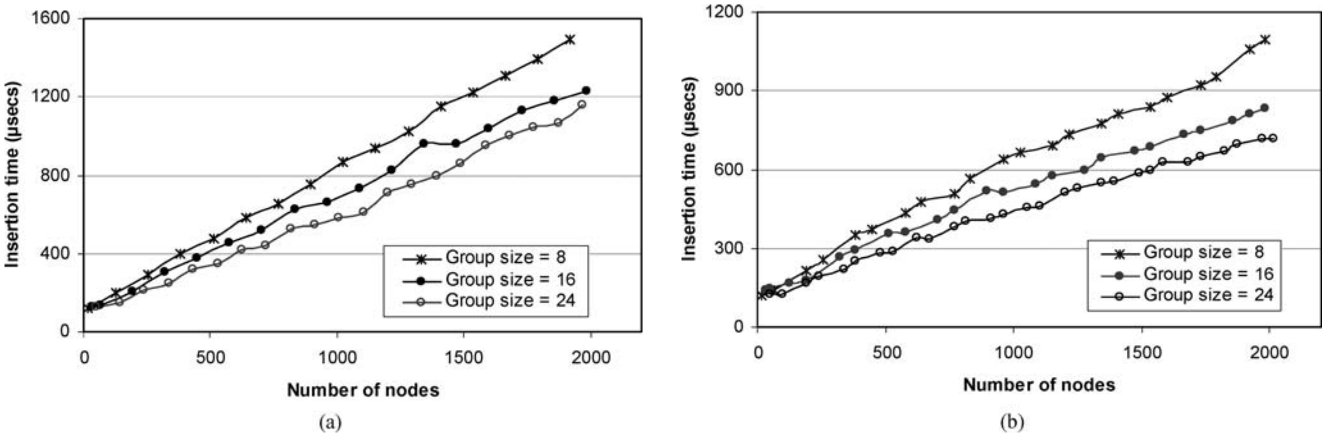


Figure 18. Node-insertion time for various system size and group size with node inserted into (a) first group of the system and (b) last group of the system.

From figures 18(a) and (b), it can also be seen that node insertion time is higher when a node is inserted into the last group as compared to that when a node is inserted into the first group. Changes in data structure and memory reallocation involved with global lists take up a major part of the insertion time, as do those related to groups when the number of groups outnumber the group size for large systems. A node inserted into the first group requires large modifications to the group structures and global lists like the global live list and global name list, increasing the insertion time. Inserting a node into the last group would require only minor changes, and only in the group structure.

5.5. Optimizing GEMS

This section presents an analytical model for GEMS which will enable resource utilization projections for system sizes not available in the testbed. The projections and analytical model are subsequently used to determine the optimum configuration and setup of the service for use in large, real-world systems.

5.5.1. Analytical formula

We have extended the formulae provided in [12] to model a service using an arbitrary number of layers. The parameters involved in the model are described below, where B is the bandwidth requirement per node,L_k is the length of the kth-layer gossip packet, λ is the number of layers used in the service, f_k is the frequency of the kth-layer gossip message, M_k is the size of monitored performance data in layer k, and g_k is the number of nodes or groups in layer k that together form a group in layer $k{+}1$; i.e.,g_k is the size of a group at layer $k{+}1$ ($e.g.$, g_1 is L2 group size).

The bandwidth requirement per node (B) is a function of the gossip message size (L) and the message transmission frequency (f) as described by Eq. (1). A gossip message includes a gossip header, a bit-set vector, gossip lists and suspect matrix as mentioned in Section 3 along with monitored performance data. The gossip header field is 4 bytes in size and it specifies the type of the gossip packet and the length of the packet. Assuming the gossip communication is in the jth layer, the bit-set vector is a bit vector whose ith bit is set to '1' if the ith group in the jth layer is alive, otherwise it is set to '0'. The length of the bit-set vector field is adjusted to an integral multiple of 8 to fit the byte-oriented structure of the UDP packet. The gossip list field is a sequence of bytes, with each byte containing 'heartbeat' data for each group in the jth layer. The suspect matrix field contains the jth-layer suspect matrix encoded into a bit sequence. The length of the suspect matrix is also adjusted to fit into an integer number of bytes. However, the physical length of the gossip packet in the transmission frame is obtained by adding the overhead contributed by the UDP and Ethernet protocols (42 bytes) to the payload length as shown in Eq. (2).

The number of gossip lists and suspect matrices depends upon the layer in which the gossip message is communicated. For example, a L1 gossip message would include the gossip lists and suspect matrices of all the layers above L1 while a L2 gossip message would include these data structures for all the layers above L2. The same description is applied in Eq. 2 to represent the size of a L1 gossip message in a λ-layered system.

$$B = \sum_{i=1}^{\lambda} L_i \times f_i \tag{1}$$

$$\begin{aligned} L_1 &= 42 + 4 + \sum_{k=1}^{\lambda} \left[\left\lceil \frac{g_k}{8} \right\rceil + g_k + g_k \times \left\lceil \frac{g_k}{8} \right\rceil + M_k \right] \\ &= 46 - \lambda + \sum_{k=1}^{\lambda} \left[\left(g_k + 1 \right) \left(\left\lceil \frac{g_k}{8} \right\rceil + 1 \right) + M_k \right] \end{aligned} \tag{2}$$

In systems that rely only on broadcast to intimate failures and do not append the local live lists of groups to the messages between groups, inter-group gossip messages are similar to those messages that are exchanged between nodes in L1 but with fewer gossip lists and suspect matrices. In such systems, the size of gossip messages communicated in any layer above L1 is calculated using Eq. (3).

$$\begin{aligned} L_j &= 46 - (\lambda - j + 1) + \sum_{k=j}^{\lambda} \left[(g_k + 1) \left(\left\lceil \frac{g_k}{8} \right\rceil + 1 \right) + M_k \right] \\ &\quad 2 \le j \le \lambda \end{aligned} \tag{3}$$

In systems that append the local live list of groups to inter-group messages, the size of a gossip message communicated in any layer above L1 is calculated using Eq. 4. The size of every inter-group message is now increased by a factor dependent on the size of the local live list which in turn depends on the number of nodes in the group (g_1). The length of the live list field is adjusted to an integral multiple of 8 to fit the byte-oriented structure of the UDP packet.

$$\begin{aligned} L_j &= 46 - (\lambda - j + 1) + \left\lceil \frac{g_1}{8} \right\rceil \\ &\quad + \sum_{k=j}^{\lambda} \left[(g_k + 1) \left(\left\lceil \frac{g_k}{8} \right\rceil + 1 \right) + M_k \right] \quad 2 \le j \le \lambda \end{aligned} \tag{4}$$

5.5.2. Optimizing system configuration

The formula to calculate the bandwidth can be used to optimally configure the service for a given system size. The number of layers and the group size at each layer required to achieve minimum bandwidth utilization may be calculated for any system size. However, this effort requires the formula be differentiated to find the minima, but an equation with a ceiling function is discrete and not easily differentiable. As such, it is not practical to provide a single formula which produces an optimum system configuration. Instead, using Matlab, we determined the optimum group size given the system size and number of layers by calculating the bandwidth required for each possible group size and selecting the value which results in minimum bandwidth utilization. For simplicity, we assume that no performance data is measured and thus M_k is set to zero. From the patterns followed by the results, we developed generalizations (heuristics) for calculating the group size which result in minimum bandwidth for a system with λ layers and a total of n nodes. Listed below is the algorithm to determine the optimum configuration.

For systems with two or more layers:

1. If $\lambda = 2$ then
   a. Set x to $n^{1/2}$
   b. Set g_1 to the multiple of eight closest to x
   c. If two multiples of eight are equidistant from x, select the smaller one

   e.g. n =144 $\Rightarrow x$ =12 $\Rightarrow g_1 = 8$ L1 nodes per L2 group
2. If $\lambda = 3$ then
   a. Set x to $n^{1/3}$
   b. Set g_1 to the smallest multiple of eight that equals or exceeds x
      e.g. n =144 $\Rightarrow x$ =12 $\Rightarrow g_1 = 16$ L1 nodes per L2 group
   c. Set g_2 to $(n \div g_1)^{1/2}$ rounded to the nearest integer
      e.g. n =512 $\Rightarrow g_1$ = 8 L1 nodes per L2 group,
      $g_2 = 8$ L2 groups per L3 group
3. If $\lambda = 4$ then
   a. Set x to $n^{1/4}$
   b. Set g_1 to the smallest multiple of eight that equals or exceeds x
   c. Set g_2 to $(n \div g_1)^{1/3}$ rounded to the nearest integer
      e.g. n =512 $\Rightarrow g_1 = 8$, $g_2 = 4$
   d. Set g_3 to $(n\div (g_1 \times g_2))^{1/2}$ rounded to the nearest integer
      e.g. n =512 $\Rightarrow g_1 = 8$, $g_2 = 4$, $g_3 = 4$
4. In general,
   a. Set x to $n^{1/\lambda}$
   b. Set g_1 to the smallest multiple of eight that equals or exceeds x
   c. Set g_2 to $(n \div g_1)^{1/(\lambda-1)}$ rounded to the nearest integer
   d. Set g_3 to $(n \div (g_1 \times g_2))^{1/(\lambda-2)}$ rounded to the nearest integer
   e. Set g_4 to $(n\div (g_1 \times g_2 \times g_3))^{1/(\lambda-3)}$ rounded to the nearest integer
   f. ...

As an example to illustrate the algorithm, consider a system with 4096 nodes. Assuming that the system is divided into 4 layers, g_1 should be set to 8 (smallest multiple of 8 that equals or exceeds $4096^{(1/4)} = 8$), g_2 should be set to 8 $\{(4096\div 8)^{(1/3)} = 8\}$ and g_3 should be set to 8 $\{(4096 \div (8 \times 8))^{1/2} = 8\}$.

Figure 19 shows the bandwidth requirement per node calculated for a system configuration based on both the heuristic and actual minima calculated using the Matlab. The results validate the heuristic, as bandwidth calculations based on generalized group sizes closely match the actual minima. The figure also illustrates the bandwidth overhead for two-layered and three-layered services for systems with fewer than 400 nodes. For systems with 64 or more nodes, a three-layered service is better than a two-layered service in terms of bandwidth overhead.

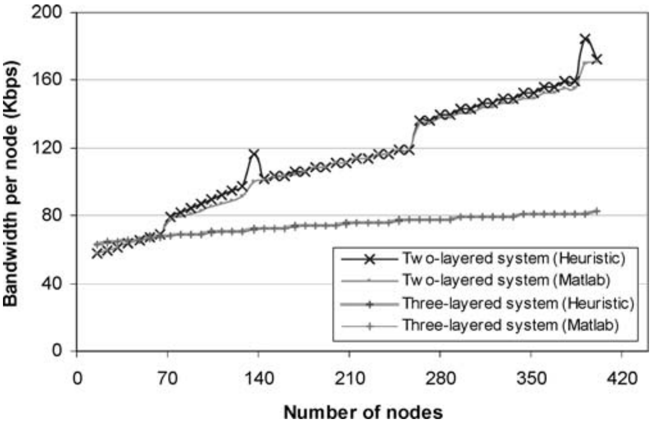


Figure 19. Verification of the heuristic for determining minimum bandwidth utilization.

Unlike the exhaustive Matlab approach, the heuristic may be easily used to determine the bandwidth requirements per node for very large systems with various numbers of layers. Figure 20 illustrates the minimum bandwidth for systems up to 6,000 nodes with different numbers of layers. These results may be used to determine the optimum configuration to achieve the best performance with minimum cost in terms of resource utilization. Based on these results, Table 4. specifies the number of layers (λ) that should be used to achieve minimum bandwidth utilization for various ranges of system

Table 4
Optimum system configuration for minimum resource utilization.

System size			
Lower bound	Upper bound	Number of layers	L2 group size (g_1)
8	63	2	8
64	511	3	8
512	4,095	4	8
4,096	32,767	5	8

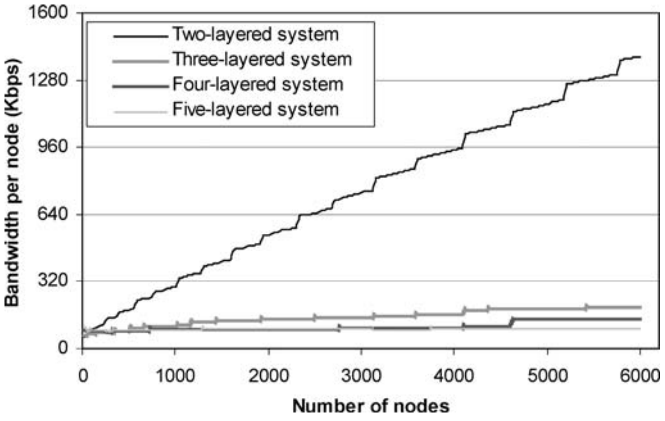


Figure 20. Projection of minimum bandwidth requirement per node for large systems.

sizes. With λ fixed, the number of groups that the system should have in each layer can then be determined using the algorithm listed earlier.

Projections also demonstrate improvement in network utilization provided by a multilayered gossip service. For example, the minimum bandwidth consumption per node in a 6,000-node system is 1392 Kbps with a two-layered service while it is just 101 Kbps with a five-layered service.

Although not shown here, CPU utilization measurements have been observed to follow the same pattern as network utilization measurements. It stands to reason that CPU utilization should scale with network utilization, as the CPU must perform work whenever a gossip packet is transmitted or received. Therefore, minimum network utilization breeds minimum CPU utilization, and the general rules of figure 19 yield minimum overall resource utilization. The service developed here has thus been demonstrated to be scalable to systems of any arbitrarily large size.

6. Conclusions

In this paper, we have presented an efficient and scalable gossip-style resource monitoring service to address the needs of high-performance applications and high-availability systems. The gossip-based dissemination of the resource monitoring data including liveness and performance information is highly robust providing fault tolerance and reduced overhead. Data consistency is also maintained, at no extra cost using the heartbeat mechanism. Scalability is greatly improved by hierarchical design of the service, confining local data within the group while providing aggregate data outside the group. The service is extensible, with provisions for adding new sensors and disseminating application-specific data. We also provide provisions for dynamic inclusion of new aggregation functions, helping applications in generating a customized aggregate view of each group. Finally, a node-insertion mechanism has been embedded into the service to enable dynamic system reconfiguration.

The performance of GEMS was comprehensively analyzed in terms of resource utilization and failure detection times on a cluster testbed of 150 nodes. The efficiency of the node-insertion mechanism has been analyzed based on experiments measuring and projecting node-insertion delay. Performance projections and optimal system configuration have been presented through an analytical model to promote efficient use of the service in real-world systems of any arbitrary size including terascale clusters.

Increasing the number of layers with system size increases the efficiency of the service. Optimum use of the service is achieved when a flat service is used for systems with fewer than eight nodes, a two-layered service is used for systems with fewer than 64 nodes, a three-layered service for those with fewer than 512 nodes, a four-layered for those with fewer than 4,096 nodes, and a five-layered for those with fewer than 32,768 nodes. Failure detection time can be as low as 130 ms for systems of any size with a L2 group size of eight. Per-node bandwidth used by the service can be kept as low as 101 Kbps even for systems as large as 25,000 nodes with a five-layered service.

Directions for future research include the investigation of issues that must be solved to move the service from clusters to grids, including security and policy management. The dynamic reconfiguration facilities provided also need to be improved to provide a plug-and-play kind of system, which implies that the system can be reconfigured at the request of the application using the service. Another direction of interest is a feasibility analysis on how best to couple GEMS with application middleware such as MPI or PVM for cluster computing. Services such as dynamic load balancing and scheduling can be provided over GEMS, with an analysis to study the improvement over presently existing mechanisms. 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. Another interesting research topic would be to use GEMS in virtual computing to monitor virtual resources. Presently, GEMS is being adapted to monitor complex heterogeneous hardware resources in support of FPGA-based Reconfigurable Computing (RC) clusters. As part of this effort, GEMS is being integrated with CARMA (Comprehensive Approach to Reconfigurable Management Architecture), an RC cluster management framework being developed at the University of Florida. GEMS is also being integrated with MonALISA, a grid-monitoring service developed by Caltech and CERN to efficiently monitor grids comprised of thousands of nodes across multiple computational sites. Finally, GEMS is being augmented with sensors to measure network parameters such as bandwidth and latency to build a robust, scalable network monitoring service.

References

[1] R. Wolski, Dynamically forecasting network performance to support dynamic scheduling using the network weather service, Cluster Computing, 1 (1) (1998) 119–131.

[2] 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, 15 (5/6) (1999) 757–768 .

[3] Z. Liang, Y. Sun, and C. Wang, Clusterprobe: An open, flexible and scalable cluster monitoring tool, in:*Proceedings of 1st* IEEE Computer Society International Workshop on Cluster Computing, Melbourne, Australia, (1999) 261–268.

[4] R. Buyya, PARMON: A portable and scalable monitoring system for clusters, International Journal on Software: Practice & Experience, 30 (7) (2000) 723–739.

[5] R. Van Renesse, K. Birman, and W. Vogels, Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining, ACM Transactions on Computer Systems 21 (3) (2003).

[6] International Business Machines Corporation, IBM LoadLeveler: User's Guide (September, 1993).

[7] J. Basney and M. Livny, Managing network resources in condor, in:*Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburgh, Pennsylvania (2000) pp. 298–299.

[8] R. Van Renesse, R. Minsky and M. Hayden, A gossip-style failure detection service, in: *Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing Middleware*, England, (1998) pp. 55–70.

[9] M. Burns, A. George, and B. Wallace, Simulative performance analysis of gossip failure detection for scalable distributed systems, Cluster Computing, 2 (3) (1999) 207–217.

[10] S. Ranganathan, A. George, R. Todd, and M. Chidester, Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters, Cluster Computing, 4 (3) (2001) 197–209.

[11] K. Sistla, A. George, R. Todd and R. Tilak, Performance analysis of flat and layered gossip services for failure detection and consensus in scalable heterogeneous clusters, in: *Proc. of IEEE Heterogeneous Computing Workshop at IPDPS*, San Francisco, CA, (2001) pp. 23–27.

[12] K. Sistla, A. George and R. Todd, experimental analysis of a gossip-based service for scalable, distributed failure detection and consensus, Cluster Computing, 6 (3) (2003) 237–251.

[13] W. Vogels, D. Dumitriu, A. Agarwal, T. Chia and K. Guo, Scalability of microsoft cluster service, in: *Proceedings of the 2nd USENIX Windows NT Symposium*, Seattle, Washington, August 3–4 (1998).

[14] H. C. Lin and C. S. Raghavendra, A dynamic load balancing policy with a central job dispatcher (LBC), IEEE Transactions on Software Engineering 18 (2) (1992) 148–158.

[15] S. Zhou, A trace-driven simulation study of dynamic load balancing, IEEE Transactions on Software Engineering 14 (9) (1988) 1327–1341.

[16] M. Zaki, W. Li and S. Parthasarathy, Customized dynamic load balancing for a network of workstations, Journal of Parallel and Distributed Computing 43 (2) (1997) 156–162.

[17] M. Willebeek-LeMair and A. Reeves, Strategies for dynamic load balancing on highly parallel computers, IEEE Transactions on Parallel and Distributed Systems 4 (9) (1993) 979–993.

[18] C. Xu, B. Monien, and R. Luling, Nearest neighbor algorithms for load balancing in parallel computers, Concurrency: Practice and Experience 7 (7) (1995) 707–736.

[19] I. Ahmed, Semi-distributed load balancing for massively parallel multicomputer systems, IEEE Transactions on Software Engineering, 17 (10) (1991) 987–1004.