# An Efficient Topology-Adaptive Membership Protocol for Large-Scale Network Services

Lingkun Chu Jingyu Zhou Tao Yang

{lkchu, jzhou, tyang}@cs.ucsb.edu
Ask Jeeves Inc., Piscataway, NJ 08854
Department of Computer Science, University of California at Santa Barbara, CA 93106

## Abstract

A highly available large-scale service cluster often requires the system to discover new nodes and identify failed nodes quickly in order to handle a high volume of traffic. Determining node membership efficiently in such an environment is critical to location-transparent service invocation, load balancing and failure shielding. In this paper, we present a topology-aware hierarchical membership service which divides the entire cluster into membership groups based on the network topology among nodes so that the liveness of a node within each group is published to others in a highly efficient manner. Our design also supports the deployment of service clusters in multiple data centers through membership proxies which update membership information incrementally so that membership communication across data centers is minimized as much as possible. We compare our approach with other two alternatives: an all-to-all multicasting approach and a gossip based approach. Our evaluation shows that the proposed topology-aware approach is scalable and effective in terms of high membership accuracy, short view convergence time, and low communication cost.

## 1 Introduction

Many Internet services are hosted in large-scale clusters with thousands of machines. Component and network failures in such an environment are common and occur frequently due to hardware failures, software bugs and operational errors [17]. The cluster configuration can also change dynamically when the service evolves, for example, new nodes can be added into the the cluster for service replacement, upgrade or expansion. To make the service available all the times, multiple hosting centers are often deployed across countries and traffic is shifted among them based on locality, load, availability and operational needs. Service components inside a cluster often rely on a membership protocol to learn what services are available locally in a hosting center or among hosting centers. In this way, internal and external service invocations can be effectively executed by knowing the availability in advance. The role of a membership service is to maintain a yellow page directory of available services hosted on all cluster nodes. When a consumer node plans to acquire a certain service, it looks up the yellow page directory and gets a set of appropriate nodes which can serve the request. Component availability information provided by the membership service should be as complete and accurate as possible to avoid sending requests to a non-functioning node. Once the information of available nodes is obtained, a consumer node in a system can make a well-informed decision, for example, load balancing based on the random polling technique [20]. Since the scale of an Internet service in a large site can grow quickly, it is important to develop a membership protocol which is incrementally scalable from a small cluster to a large-scale cluster with thousands of nodes hosted in multiple data centers [2].

The architecture of a membership service can be centralized, where a stand-alone server provides membership information to all the service nodes, or distributed, where every consumer maintains its own yellow page directory. Examples of the centralized approach can be found in distributed file systems, such as Google FS [10], where the membership service is often integrated with the directory service. Although the centralized approach is simple to implement, it is not scalable and the membership server is a single point of failure. Furthermore, it introduces an additional delay during a service invocation since the membership server needs to be contacted first to look up the actual server node. Such overhead is not acceptable in a web site where low response time under a high volume of traffic is desired.

In this paper, we focus on a decentralized approach for large-scale service clusters [20, 21]. In this approach, membership information is made available to every node that needs to invoke external services and this information is maintained as soft state to minimize management overhead and data inconsistency [18]. More specifically, each node is able to access entire yellow page directory inside a service cluster. This simplifies management and makes the service easy to extend by allowing a node to seek services from any other nodes. In addition to the aliveness information, our membership service also maintains relative stable information of each node, such as application service name, partition ID, machine configuration,

* This work was supported in part by Ask Jeeves, NSF grants CCF-0234346, ACIR-0086061.

etc. Dynamic information such as workload is not covered by the membership protocol itself. Instead, external protocols, such as random polling load balancing scheme can be built on the top of our membership service.

The proposed approach considers the following requirements. First, the design should minimize communication and computation overhead of maintaining membership information at each node. Secondly, the scheme should be scalable to thousands of nodes in a single service cluster and to multiple data hosting centers. Thirdly, the membership service needs to be complete, accurate and responsive. It should detect any status change, i.e., node departures and joins (*complete*), and the detection should be correct (*accurate*) and as quick as possible (*responsive*). Previous research has not provided a complete solution to meet all of the above requirements. Directory-based membership services studied in [5, 12] only focus on peer-to-peer environments in wide area networks or a small-scale cluster. The requirement for low communication overhead for high traffic web sites is more demanding than previous studies.

In this paper, we propose a topology-aware membership scheme for large-scale clusters to meet the above requirements. Since a large-scale cluster is often grouped through a layer-3 switch, our scheme automatically divides cluster nodes into subgroups based on the communication layer setting by the system administrator. A leader is elected for each group from time to time when a failure occurs. Group leaders form a hierarchical tree structure to exchange membership information across groups. Across multiple data centers, a membership proxy protocol is developed to distribute membership information and to enable service invocation from one data center to another.

The rest of the paper is organized as follows. Section 2 presents the background of the membership service in clustering middleware. Section 3 describes our protocol for membership service and an extended version for multiple data centers. Section 4 analyzes the scalability of our approach with a comparison to two alternatives. Section 5 discusses our membership service API. Section 6 presents the details of the implementation and our evaluation in a cluster with hundreds of nodes. Section 7 discusses related work. Section 8 recaps our findings and concludes the paper.

## 2 Background

In this paper, we propose a membership service for cluster-based Internet services. The approach is implemented in the *Neptune* framework – programming and runtime support for building cluster-based Internet services [21]. Additionally, it can be easily coupled into other clustering frameworks.

Neptune targets cluster-based network services accessible to Internet users. Requests issued by remote clients enter service clusters through protocol gateways such as Web servers or XML gateways. Inside the service cluster, services are typically composed of several service components. Persistent data for service components are usually partitioned and replicated for incremental scalability and high availability. We use the term *Service Instance* to denote a server entity that runs on a cluster node and manages a data partition belonging to a service component. Neptune employs a functionally symmetric and decentralized clustering design. Each service instance can elect to provide services (when it is called *service provider*) and it can also acquire services exported by other service instances (when it is called *service consumer*). This model allows multi-tier or nested service architecture to be easily constructed.

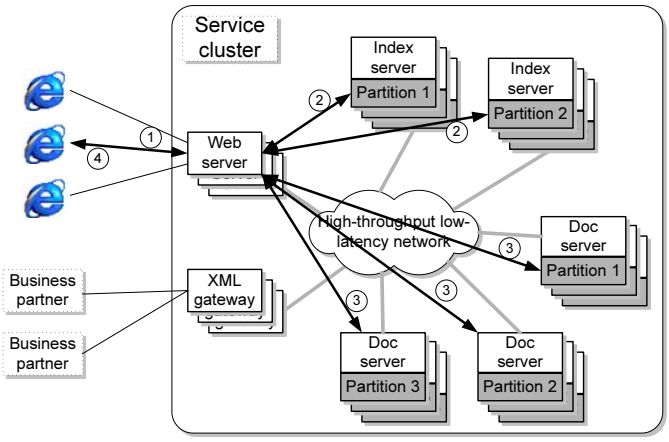


Figure 1: A prototype search engine service supported by Neptune

Figure 1 illustrates the architecture of a prototype document search service supported by the Neptune middleware. In this example, the service cluster delivers a search service to Internet users and business partners through Web servers and XML gateways. Inside the cluster, the main search task is decomposed into two parts and distributed to index servers and document servers. The data for both components is partitioned and replicated. In Figure 1, there are two index server partitions and three document server partitions. Each partition has three replicas. The arcs labeled with ①–④ in Figure 1 show a simplified work flow of serving a client request. ① A search query arrives at one of the protocol gateways. ② The protocol gateway contacts the index server partitions to retrieve the identifications of documents relevant to the search query. ③ The protocol gateway contacts the document server partitions which translate the list of document identifications to human understandable descriptions. ④ Finally, the protocol gateway compiles the final results in HTML or XML format and returns them back to the client. In the above work flow, the protocol gateway contacts the service instances through the *Neptune consumer module*. On the service provider side, the requests are received by the *Neptune provider module*, which subsequently invokes the service-specific handlers to process the requests.

Among the important features that Neptune needs to provide in the middleware layer, there are: (1) transparent service location and (2) failure isolation. Service component partitions are

addressed through location-transparent names (service name, partition ID). The Neptune consumer module automatically routes each request to an appropriate node based on the service availability and runtime workload. The Neptune provider module at each node periodically announces a service availability, or heartbeat message to other nodes. Faulty nodes will be automatically detected by the discontinuance of heartbeat messages. This is achieved through soft state techniques [18]. These two features are directly related to the Neptune membership service.

One straightforward approach of the membership service is to let every node periodically send its heartbeats to other nodes and collect heartbeats from other nodes. This is the all-to-all approach. Each heartbeat packet contains service information and current load status of a node. Every node builds its own membership directory based on these heartbeat packets. This is a fully distributed approach in the sense that every node maintains its membership directory independently. Neptune adopts this scheme when a service cluster is in a small scale. The advantage of this approach is that each node functions independently and it provides the best fault isolation. Unfortunately, this simple scheme is not scalable. The communication and computation overhead in maintaining a local yellow page directory can be significant when a cluster has thousands of nodes as shown in Figure 2. The measurement is done on a Linux machine with dual 1.4 GHz P-III processors. We vary the number of heartbeat packets that received by the machine to emulate the expansion of the cluster. If each node sends a 1024-byte heartbeat packet per second, the heartbeat packets can consume 4MB/s bandwidth for a cluster with 4000 nodes, which is 32% of the raw bandwidth of a Fast Ethernet link.

Another alternative approach widely used in wide-area network applications is gossip style membership service [23]. In a gossip style approach, each node randomly picks up a set of neighbor nodes and sends them its current membership directory. The recipient nodes then update their own directories based on the new information. Given an error tolerance, this approach can control the amount of heartbeat traffic by selecting an appropriate number of neighbor nodes. This capability is essential in wide-area services where bandwidth is limited, network latency is high and multicast is not available. When it comes to low latency and high bandwidth system area networks, a gossip style membership services has the problem of slow convergence time and high communication overhead for maintaining a complete view of the service cluster. Furthermore, its probabilistic property does not guarantee 100% accuracy, which can be unacceptable for some high reliable services.

The purpose of the Time-To-Live (TTL) field in an IP packet header is to prevent packets falling into infinite routing loops, which can occur when due to misconfiguration, instability during routing repair and etc. The TTL field puts an upper limit on these looping packets. It indicates the maximum number of hops a packet may transit. When an IP packet passes a router, the router decrease the TTL by one. Then the count reaches zero, the packet will be discarded and an ICMP ”time exceeded” message will be sent back to the originator of the packet. The network tool, *traceroute*, uses this feature to discover the routing path from a host to another. Then the network topology can be derived from these routing pathes.

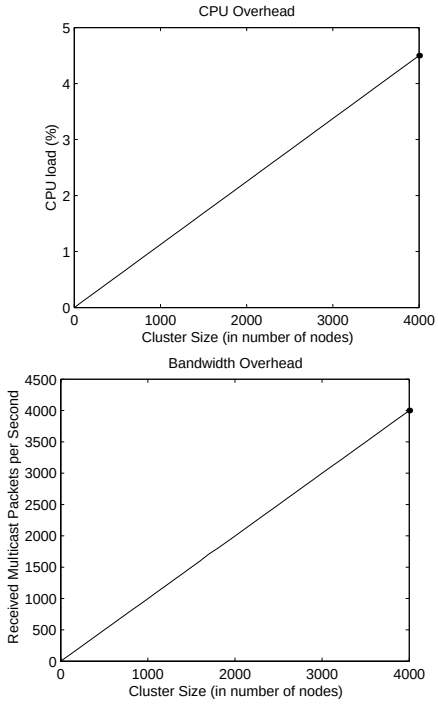


Figure 2: All-to-all approach is not scalable.

In this paper, we use this feature to limit the scope of multicast packets so that we can form multicast groups based on the network topology, which can automatic adapt to the network topology and effectively conform the multicast traffic to the network topology.

# 3 Hierarchical Membership Service

This section presents the details of our proposed hierarchical membership service for large scale service clusters. Figure 3 shows an overview of the hierarchical membership service. It contains two parts: (1) a tree-based membership protocol which deals with a large scale service cluster hosted in one data center, (2) a membership proxy protocol which coordinates service invocation across multiple data centers. Now we illustrate the details of these protocols.

## 3.1 Tree-based Membership Protocol

The tree-based membership protocol deals with a large scale service cluster hosted in one data center. It exploits the network topology of a cluster to build a hierarchy of membership groups, which can greatly reduce network traffic in maintaining the membership service. The basic idea of this approach is to automatically form small size multicast groups based on

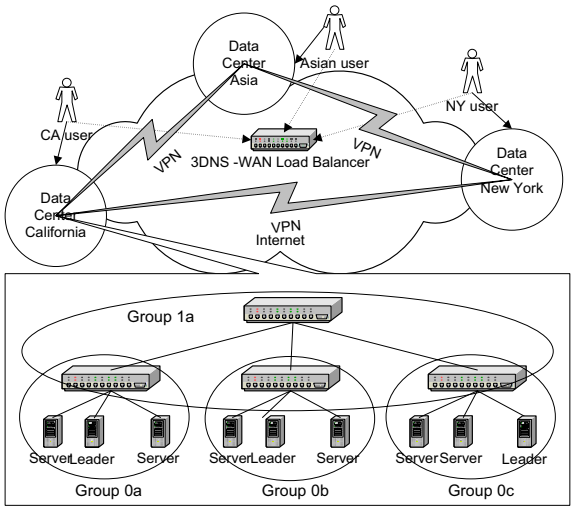


Figure 3: A Hierarchical Membership Service.

network topology using TTL values. Inside each group, nodes periodically send heartbeat packets to each other. A group leader is elected to detect local status changes and to relay the changes. Lower level group leaders join a higher level group and elect a group leader, which in turn can join an even higher level group. Following this joining process, a tree is finally formed as shown in Figure 3 in the California data center.

### 3.1.1 Group Management

Group management is essential to the tree-based membership service. Its main functions include topology-aware group formation, failure detection, and group leader election.

- **Topology-aware Group Formation.** The construction of hierarchical groups is based on the network topology of a cluster. The idea is to form groups with nodes that are close to each other in each group. We measure the distance between two nodes as the TTL count between them. Each group is associated with a TTL value. We assume each node uses only one IP address to join membership groups. We first present how the formation scheme works when the network layout fits a specific tree topology where each host or router has only one uplink and all hosts are at the same level from the root router. Then we extend the scheme to general network topologies.

  **Tree topology.** In this topology, TTL counts fit into a transitive relation. If there is $t$ TTL hops from node $A$ to node $B$ and from node $B$ to node $C$, node $A$ can also reach node $C$ within $t$ hops. For example when the TTL value of a group is one, all nodes in the group are at most one hop from each other. In terms of physical topology, nodes in this type of groups are connected by a layer-two device or within the same VLAN. We call this type of groups as level 0 groups or local groups. Level 0 group leaders join a multicast channel with the TTL value set to two. This forms level 1 groups and their leaders can join higher level groups subsequently. This process continues until a tree is formed. In practice, we set a limit on the maximum TTL value which is no less than the longest path between two nodes in the cluster. When the maximum TTL value is reached, the group formation process stops. We assign different multicast channels for groups at different levels. Groups at the same level can share the same multicast channel since packets from a group will not reach another group at the same level because of the TTL limit. Otherwise, these two groups should be merged into one since all the nodes inside the group can be reached using the same TTL value. Only a base multicast channel needs to be specified for a cluster. All other channels can be derived from the base channel and a TTL value. This not only reduces the number of multicast channels required by the membership service but also simplifies the administration task. Without the help of topology-aware group formation, an administrator first needs to get the complete knowledge of the network layout, manually divides nodes into groups, and then assigns multicast channels to these groups. Any expansion of the cluster can undo the previous configuration and make the cluster hard to manage. For maximum control flexibility, our implementation also allows administrators to specify multicast channels at each level.

  **Other topologies.** Now we extend the above scheme to general cases. We still use TTL values to form groups. The main difference from the above topology is that groups at the same level can be overlapped. In other words, some nodes can belong to multiple groups at the same level. This is because transitive relation of TTL counts may not hold in a general topology. Figure 4 shows an example of such topologies. Node $A$, $B$ and $C$ are leaders for the level 0 groups. Node $B$ can reach Node $A$ and Node $C$ within 3 hops. But Node $A$ and Node $C$ need 4 hops to reach each other. Note that level 1 groups are skipped as each group only has one member. There are two possibilities when electing group leaders for the level 2 groups $2a$ and $2b$. Node $B$ can be the leader for the both groups, or Node $A$ and Node $C$ can be the leader for Group $2a$ and Group $2b$ respectively. Node $A$ and Node $B$ cannot be the leaders at the same time since our group leader election algorithm guarantees that a group leader cannot see other leaders at the same level. In the former case that Node $B$ is the leader for the both groups, the higher level group sees Group $2a$ and Group $2b$ as one group represented by Node $B$. In the latter case, two leaders, Node $A$ and Node $C$ both join the higher level group. Thus Group $2a$ and Group $2b$ are deemed as separate groups at the higher level. The two cases make no difference in the sense that group leaders can always propagate membership information from or to all group members. Furthermore, because the operation caused by an update message at each node is idempotent, redundant

messages will not cause confusion.

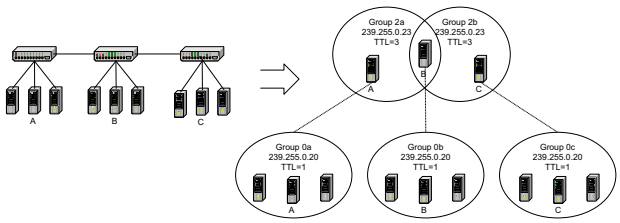


Figure 4: Group overlapping at the same level.(Left) Physical network layout. (Right) membership groups.

- **Failure Detection.** Since we use multicast to disseminate heartbeat packets, it is possible these packets can be lost due to network congestion or overloading senders or receivers. Thus, we declare a node is dead when no heartbeat packet is received from the node after a pre-defined time period, which can be chosen when the probability of multiple consecutive packet losses during the period is negligible. Each node in a group performs the failure detection independently. After the group leader detects a node failure, it will multicast this information to all the groups that it joins. For example, if node $B$ is the group leader for both groups in Figure 4 and it detects node $A$ fails, node $B$ will multicast this failure in the both level 0 and level 2 groups so that node $C$ also knows of the failure and notifies other nodes in group $0c$. Similarly, a group leader will also inform all other groups when a new node joins.

- **Leader Election.** Each group has a group leader and a backup leader. The backup leader is randomly chosen by the primary group leader and it will take over the leadership if the primary leader fails. This allows quick recovery if only the primary leader fails. When both the primary and the backup leader fail, an election algorithm is used to select a new primary leader, which will designate a backup leader. The leader election is performed using the bully algorithm [4]. Each node is assigned a unique ID (e.g., IP address). The member with the lowest ID becomes the group leader. If there is already a group leader, a node will not participate the leader election in any groups with the same multicast address and TTL value. For example, if node $A$ is already the leader for group $2a$ in Figure 4, node $B$ will not participate in the leader election for group $2b$ although it may have a lower ID than node $C$. Eventually, node $C$ will become the leader of group $2b$.

### 3.1.2 Membership Maintenance Protocol

The membership maintenance protocol ensures that every cluster node has a local copy of the global service directory and it is complete and accurate. Several sub-protocols are used to achieve this goal.

- **Bootstrap Protocol.** This protocol allows a newly joined node to quickly build its local yellow-page directory. When a node joins a group, it first listens to the multicast channel to collect heartbeat packets from other group members. A group leader is found if a special flag in its heartbeat packets is set. Then the new node contacts the leader to retrieve the membership information that the leader knows. Meanwhile, the group leader also asks the new node for the membership information that it is aware of in case that the new node is also a group leader from a lower level group. The result is then propagated to all group members using the following update protocol.

- **Update Protocol.** This protocol determines how an update message is propagated to all nodes in a cluster. When a level 0 group leader detects a change (a node departure or a node join), it sends out an update message to the level 1 group it joins. Other nodes in the level 0 group can detect the failure by themselves. The leader of the level 1 group will relay the message to the next higher level group. This procedure continues until no higher level group exists. At any level, other group members relay this message to the groups where they are the group leaders. In this way, the update message can be propagated to all nodes in the cluster. Figure 5 illustrates the propagation of an update message. Node $B$, $E$ and $H$ are the group leaders of the three level 0 groups respectively. Node $E$ is the group leader of the level 1 group. Node $B$ first detects Node $C$ is dead and removes it from $B$'s membership table. Node $B$ then multicasts this information to Group $1a$. Node $E$ and $H$ receive the message and propagate it to Group $0b$ and Group $0c$ respectively. Other group members receive this message and delete Node $C$ from their membership tables. Since node $E$ is the leader of Group $1a$, it also relays the message to Group $2a$.

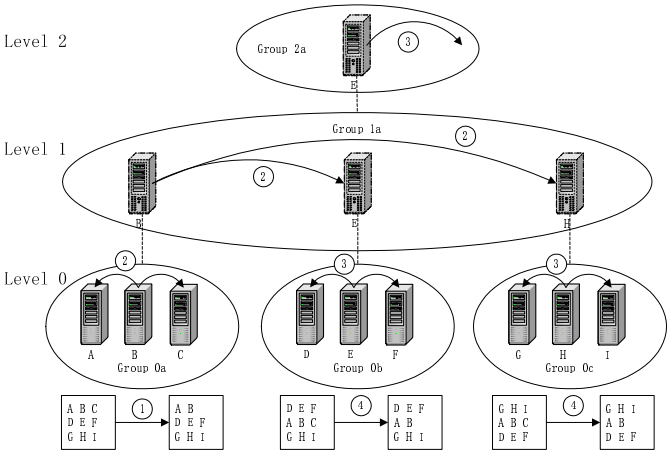


Figure 5: Propagation of an update message. The circled numbers show the propagation order of the update message.

- **Timeout Protocol.** This protocol is used to remove stale entries in the yellow-page directory. It works together

with the previous discussed failure detection mechanism. For all nodes that send heartbeat packets to node $N$, $N$ purges any of these nodes from its membership directory after not receiving the corresponding heartbeat packets for a certain period. If a dead node is detected at a level greater than 0, membership information that is relayed by the dead node is also timeouted. This means the membership information relayed by a group leader has the same life time as the leader itself at each level. Hence, network partition failures (e.g., switch failures) can be quickly detected. If it is not a network partition but merely a group leader fails, the backup leader or newly elected leader will join the same group and exchange the membership information with other group members. To minimize the impact of a leader failure, we assign different timeout values for groups at different levels. Higher level groups are assigned with larger timeout values. Thus when a group leader fails, the lower level group can still have time to elect its new leader before the higher level group purges all the nodes of the lower level group.

- **Message Loss Detection.** When there is a cluster status change, update messages are generated to disseminate the information about the status change. All update messages are UDP packets. These packets can be lost during network transmission. To help detect a packet loss, each host assigns a sequence number for an update message. Thus the receiver can use the sequence number to detect lost updates. Since each update about a node departure or join is very small, we let an update message piggyback last three updates so that the receiver can tolerate up to three consecutive packet losses. If more than three consecutive packets are lost, the receiver will poll the sender to synchronize its membership directory.

## 3.2 Membership Proxy Protocol

We introduce membership service proxies in each data center to enable exchange of membership information and service invocation across multiple data centers. The proxies in a data center collect local membership information and exchange the information with proxies in other data centers.

There are multiple membership proxies for each data center to improve availability and performance. These proxies form a membership group and elect a group leader using the previously discussed tree-based protocol. One difference is that the multicast channel is predefined and the group only includes proxies. Then the group leader joins the membership channel of the service cluster to collect the membership information of the local data center. All proxies share a single external IP address using an IP failover mechanism. When the proxy leader fails, the newly elected leader will take over the IP address. Thus, all other data centers always see the same IP address and communicate with the proxy group leader. Leader proxies exchange membership information over a VPN (virtual private network). They use unicast UDP packets to communicate with each other since multicast over VPN or Internet is generally available. This will not affect scalability since the number of data center is usually limited.

When a node in the service cluster seeks a service that is not available in the local service cluster, it can contact one of the proxies to see if the service is available in other data centers. If it is available in another data center, the proxy will forward this request to one of the corresponding proxies in the remote data center. The destination proxy will then relay this request to the appropriate service node and send the results back to the original proxy. The details are shown in Figure 6: (1) a node cannot find a desired service in its local service cluster and forwards the request to one of the local proxies. (2) The proxy looks up the membership information from the other data centers and forwards the request to one of the proxies in the corresponding data center. If it cannot find an appropriate data center, the request will be rejected. (3) The proxy in the remote data center forwards the request to an appropriate backend service node. (4) The result is returned from the backend service node to the proxy. (5) The second proxy relays the result to the original proxy. (6) The original proxy returns the result to the node that issued the request.

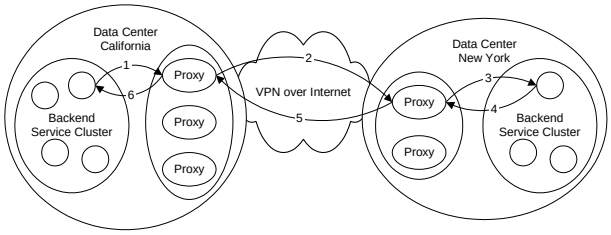


Figure 6: Service invocation through proxies.

The detailed proxy protocol works as follows.

- **Group Formation.** All proxies join a multicast channel that is reserved exclusively for them and the leader of the proxy group joins the rest of the cluster. The leader uses the same base channel as the other leaders do. It increases the TTL value to find the appropriate group to join. Thus the leader of the proxy group can appear at any level of the membership tree. It does not necessarily appear as the root of the tree.

- **IP Failover.** A proxy group leader exchanges information with other proxy group leaders in the remote data centers. When it fails, one of the remaining proxies will take over the leadership as we discussed in Section 3. Furthermore, the new leader will take over the external IP address that is exposed to the other data center. Therefore further communication from the proxy leaders in the other data centers will be directed to the new leader.

- **Heartbeat.** Each proxy leader periodically sends heartbeat packets to all other data centers. These heartbeat packets include a summary of membership information

of the local data center. Each proxy leader sends these heartbeat packets sequentially to the other leaders using well-known IP addresses. If the size of the membership summary is too big, the summary is broken into multiple heartbeat packets. When a proxy leader receives such a heartbeat packet, it relays the packet to the local proxy group through the group's multicast channel. The content of the summary of membership information can be specified through the membership service API, which will be shown in Section 6. Generally, the summary does not include the detailed machine information. It only has the availability of service information, which is much smaller.

- **Update Message.** When a proxy leader learns of a status change in its local data center and this change also updates the membership summary, the leader informs other proxy leaders immediately using UDP unicast. When other proxy leaders receive the message, they update their membership directories to reflect this change and meanwhile they relay this message to other proxies in their local data centers.

# 4 Scalability Analysis

In this section, we present a simplified analysis of the hierarchical approach with comparison to two alternatives: the all-to-all approach and the gossip approach. We assume multicast is used to disseminate messages to a group of nodes, and unicast is used for one to one communication, such as gossip messages. We also assume the total number of nodes in the cluster is $n$, the size of a membership description for each node is $m$ bytes.

We are interested in the communication cost, failure detection time, and view convergence time for all nodes. Failure detection time is how quickly a failure can be detected by other nodes. View convergence time is the interval when all nodes notice a status change after the change occurs. In the following analysis, we focus on single node failure.

There are often trade-offs between communication cost and failure detection time. We introduce a metric *bandwidth detection time product* that combines bandwidth consumption and failure detection time. Assume the bandwidth consumption of a scheme is $B$ bytes per second when the cluster status is stable, the metric is calculated as $BDP = B \times T_{fail}$, where $T_{fail}$ is the detection time of a node failure. Protocols with lower $BDP$ values are better, because they use less time to detect a failure with a fixed bandwidth.

## 4.1 Failure detection time

Let $m$, $n$, and $B$ be the average size of one node's information, the total number of nodes, and the total bandwidth, respectively. The failure detection time and $BDP$ are calculated as follows.

- **All-to-all.** The multicast frequency is limited by $f = \frac{B}{m \times n^2}$ since each node receives heartbeats from all other nodes and sends out one heartbeat per multicast cycle. Thus each node consumes $m \times n$ bandwidth per cycle. If the protocol assumes a node is dead after not hearing $p$ consecutive heartbeats from the node, the failure detection time is

$$T_{fail} = \frac{p \times m \times n^2}{B} = O(n^2), BDP = O(B \times n^2)$$

  In practice, each node often fixes its multicast frequency, which is independent of the number of nodes. This makes the failure detection time as a constant. But the total amount of network bandwidth will become $O(n^2)$.

- **Gossip.** Each gossip message contains a local view of group membership, which is $m \times n$ bytes. We do not count the broadcast message in the gossip scheme since it can be eliminated under optimization. The frequency of gossip can be calculated as $f = \frac{B}{m \times n^2}$. Its failure detection time is shown in the following formula. The detailed calculation can be found in [23].

$$T_{fail} = \frac{O(\log n)}{f} = O(n^2 \log n), BDP = O(B \times n^2 \log n)$$

  If we assume the gossip approach can consume $O(n^2)$ amount of bandwidth, the failure detection time will drop to $O(\log n)$.

- **Hierarchical.** Assume the size of each group is limited to $k$ nodes, the height of the membership tree is limited by $\log_k n$. Adding up the number of groups at each level, we get the number of total groups

$$g = \frac{n}{k} + \frac{n}{k^2} + ... + \frac{n}{k^{\log_k n}} = \frac{n-1}{k-1}$$

  The multicast frequency is $f = \frac{B}{(g \times m \times k^2)}$ since each group has $k$ nodes which consume $m \times k^2$ bandwidth. If the protocol assumes a node is dead after not hearing $p$ consecutive heartbeats from the node, the failure detection time is

$$T_{fail} = \frac{p}{f} = O(n), BDP = O(B \times n)$$

  If each node fixes its multicast frequency as the all-to-all approach does, the total amount of network bandwidth will become $O(n)$ and the failure detection time will become a constant.

In a word, the hierarchical scheme is the most scalable approach in terms of the bandwidth detection time production.

### 4.2 View convergence time

View convergence time includes the failure detection time and the time to disseminate this information to all other nodes. Similarly, we can define a metric *bandwidth convergence time product* to measure the effectiveness of the three approaches: $BCP = B \times T_{converge}$. For the Gossip and the flat scheme, the view convergence time is the same as the failure detection time since all nodes maintain their views independently. Thus, their bandwidth convergence time product are

$$BCP = O(B \times n^2)$$

and

$$BCP = O(B \times n^2 \log n)$$

respectively. For the hierarchical scheme, the view convergence time is the failure detection time plus the time to disseminate this information along the hierarchical tree whose height is $\log_k n$. An update message will first travel up to the root of the tree and propagate down to the bottom of the tree. Assume the network transmission time of an update message is $\lambda$, the whole propagation will take $2\lambda \log_k n$. Thus, the convergence time is

$$T_{converge} = T_{fail} + 2\lambda \log_k n = O(n) + O(\log_k n) = O(n), BCP = O(B \times n)$$

Again, the hierarchical scheme has the best scalability in terms of the bandwidth convergence time product.

## 5 Membership Service API

The membership service API allows all nodes to share the same configuration file to simplify the management task. The configuration file specifies some necessary parameters such as the basic membership multicast channel, multicast frequency, maximum tolerated packet losses, and the shared memory key to store the yellow page. Users can specify these parameters using an external configuration file. Figure 7 gives an example of such a configuration file. The section "*SYSTEM" specifies the global parameters as discussed before. The section "*SERVICE" specifies parameters for each service that is hosted on the machine. In this example, we have two services, a cache service and an HTTP service. Service names are specified in the brackets. The standard parameter "PARTITION" specifies the local partition number hosted on the machine. Each service can also have additional service specific parameters. For example, the HTTP service has an additional parameter "Port" to specify the port number of the service. These parameters will be sent as key-value pairs together with other membership information.

Besides the above external configuration files, users can also control the membership service using the membership service library. The library allows a service to disseminate service status information along with the membership multicast packets. The API of the membership service library is shown in Figure 8. MService is constructed from an external configuration

```
*SYSTEM
        SHM_KEY = 999
        MAX_TTL = 4
        MCAST_ADDR = 239.255.0.2
        MCAST_PORT = 10050
        MCAST_FREQ = 1
        MAX_LOSS = 5

*SERVICE
        [HTTP]
                PARTITION = 0
                Port = 8080

        [Cache]
                PARTITION = 2
```

Figure 7: An example configuration file.

file. If the configuration file is not available, default values will be used and later on they can be updated by the 'control' function. The 'run' function will create a set of threads (as shown in Figure 10 ) for the membership service. The function 'register_service' publishes a real service through the membership service. It also publishes a list of partitions that the node is in charge of. For example, a node that calls 'register_service("Retriever", "1-3")' in a search engine cluster will announce that it hosts the document retriever service for the partition 1, 2 and 3. The function 'update_value' is called when the service code wants to update its service status or other service associated values, such as a list of available service methods. Once a value is updated, the membership service will include it into its multicast packets and propagate this information to other nodes. The value can be deleted by a 'delete_value' call.

```
class MService {
        MService(const char *configuration);
        ~MService();
        void control(int cmd, void *arg);
        int run(void);
        int register_service(const char *name,
                const char *partition);
        int update_value(const char *key,
                const void *value, int size);
        int delete_value(const char *key);
};
```

Figure 8: API of the membership service library.

In order to access a yellow page directory provided by a membership service, a client program needs to link with the membership client library. The library provides safe access to the membership information collected by the daemon process. The client program can be in the same service process or another separate process. Figure 9 shows the client API. A MClient object is initialized based on a provided shared memory key, which is the communication token between the membership service and the client. When a client wants to seek a service, it calls 'lookup_service' and specifies the service name and the partition list that it desires. We support regular expressions both in the service name and the partition list. The matched nodes are stored in a MachineList. Each of its ele-

ments is a list of attributes and values that describe a machine. These attributes can include the information of the machine configuration and service configuration.

```
typedef pair<char *key, void *value> Attribute;
typedef vector<Attribute>* Machine;
typedef vector<Machine> MachineList;
class MClient {
        MClient(const char *shm_key);
        ~MClient();
        int lookup_service(const char *service,
                const char *partition,
                MachineList *machines);
};
```

Figure 9: API of the membership client library.

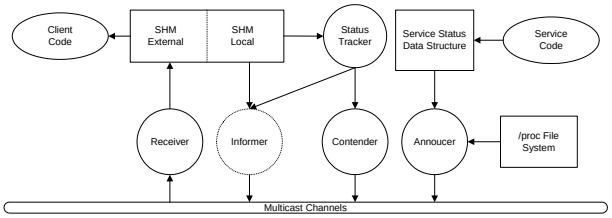


Figure 10: Components of the hierarchical membership implementation (Circles and eclipses represent active entities, rectangles represent data structures and arrows show the information fbw).

# 6 Implementation and Evaluation

In this section, we first illustrate our implementation of the hierarchical membership service. Then we evaluate it with comparison to the other two alternative approaches. The main objective is to evaluate the scalability of the hierarchical approach in terms of failure detection time, view convergence time and network overhead. Furthermore, we evaluate the effectiveness of failover using our membership proxy protocol across two data centers.

## 6.1 Implementation

We have implemented the membership service in C++ on Linux. The membership service can be run as a standalone daemon or be linked with other client code that provides the actual service. In the latter case, the actual service can publish the service configuration along with the machine configuration through the membership protocol. In our implementation, we use the random polling approach to balance the workload among replicas [20]. Therefore, we do not need to propagate load information which changes frequently. If load information is desired, an external protocol can be built on the top of our membership protocol to propagate load information. For example, the protocol can propagate load information only to interested nodes which have recently seeked the service from the service node. The details of these protocols are beyond the scope of this paper.

Figure 10 shows the components of our implementation and their relations to external entities. The components in circles are threads associated with specific tasks. The Announcer thread collects the machine information from the /proc file system and the service information from the IPC channel of the service. Then it constructs a multicast packet to include these information, and sends the packet to the multicast channels the node has joined. It also answers the polling requests from other nodes to facilitate the random polling load balancing strategy.

The Receiver thread listens to the multicast channels that the node has joined. When a packet arrives, Receiver will update the shared memory structure to reflect the new information. We use shared memory to allow fast access by service clients that may reside in different processes. The shared memory block is split into two parts: (1) a local part that contains all the nodes that a node is directly connected to via the multicast channels. (2) an external part which contains information of external groups. This information is relayed through its group leader. The difference is that a node is responsible for detecting a failure in the local part of the membership while it depends on its group leader to tell the aliveness of an external node.

The Status Tracker thread periodically checks the entries in the shared memory block and purges any expired entries. When there is expiration, it checks if the failed node is its group leader. If it is, the Tracker will assume the backup leader as the new leader. If there is no backup leader, the Tracker wakes up the Contender thread to initiate an election process for the new group leader. When there is a status change, the Tracker wakes up the Informer thread to propagate the change to other groups members if the node itself is a group leader.

Besides relaying changes to other group members, the Informer thread of a group leader also listens on a well known UDP port. Thus, the Receiver thread on a newly joined node can poll Informer to get an entire yellow page. Furthermore, the Receiver is also responsible for detecting any loss of update packets. If there is an unrecoverable loss, the Receiver will also poll the corresponding node to get a complete image.

## 6.2 Experiment settings

All the experimental evaluations were conducted on a rack-mounted Linux cluster with 100 dual 1.4 GHz Pentium III nodes. Each node runs RedHat Linux (kernel version 2.4.20). There are two Layer-3 switches with 100Mb links. One accommodates 50 nodes each. These two switches are connected by a Gigabit link.

For our hierarchical protocol, we manually designate multicast channels to emulate multiple networks. Each multicast channel hosts 20 nodes. Therefore, there are five networks for 100 nodes and these five networks form a second level network.

For Gossip scheme, mistake probability is set to 0.1%, which represents the bound that any node may make an er-

roneous failure detection. This is a relatively loose requirement for the Gossip scheme. As discussed before, each gossip message accounts for a number of network packets and the broadcast packets are not counted.

In the following experiments, we fix the multicast or gossip frequency as one packet per second for all three schemes. For the all-to-all scheme and the hierarchical scheme, we set the maximum packet losses as 5 before we declare a node is dead. We vary the number of nodes from 20 to 100 with the number of networks from 1 to 5. The average packet size carrying the membership information of each node is measured as 228 bytes for all three schemes.

## 6.3 Bandwidth Consumption

First, we compare the bandwidth consumption for three schemes in Figure 11. Bandwidth consumption is measured on each node by counting the incoming heartbeat packets. Then all numbers are added up to get the aggregated bandwidth consumption. We vary the number of nodes from 20 to 100. For the hierarchical approach, this corresponds to the number of networks from 1 to 5. When there are 20 nodes, all the schemes use the same amount of bandwidth. When the number of nodes grows, the hierarchical scheme has the least total bandwidth consumption and has close to linear growth. On the contrary, the bandwidth usage for both the all-to-all scheme and the gossip scheme grows quadratically with the number of nodes. These results are in line with our analysis results in Section 4. The average bandwidth consumption for each node remains constant in the hierarchical approach and grows linearly for the other two approaches. This suggests that the hierarchical approach is more scalable in terms of network bandwidth usage.

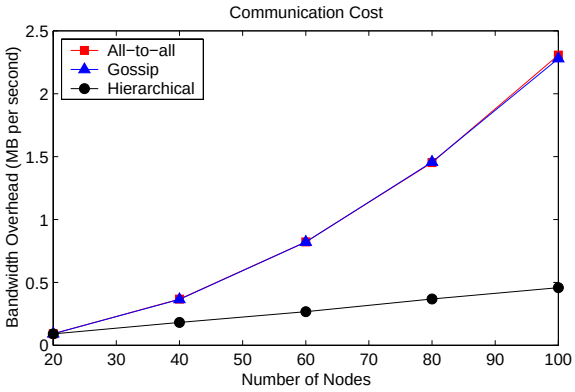


Figure 11: Bandwidth consumption

## 6.4 Failure detection time

Figure 12 shows the failure detection time for three schemes. We kill the membership service daemon process on a node to emulate the node failure. During the test, each node dumps its membership directory to a disk file when there is a change. Since the clocks on all the nodes may not be well synchronized, we send a start message to all nodes when the test begins. After the test, we find the earliest time when the failure is recorded in these log files as the failure detection time, and the latest record time of the failure as the view convergence time. When there is one network, the hierarchical scheme reduces to the all-to-all scheme. We can see from the figure that when the number of nodes grows, the hierarchal scheme and the all-to-all scheme have the same constant detection time which is roughly the maximum number of packet losses times the multicast period. On the other hand, the detection time of the gossip scheme increases logarithmically along with the number of nodes. It also has the longest detection time when there are 20 nodes. Both the hierarchical and the all-to-all schemes have similar shorter failure detection time than the gossip scheme. This experiment results also corroborate our analysis in Section 4.

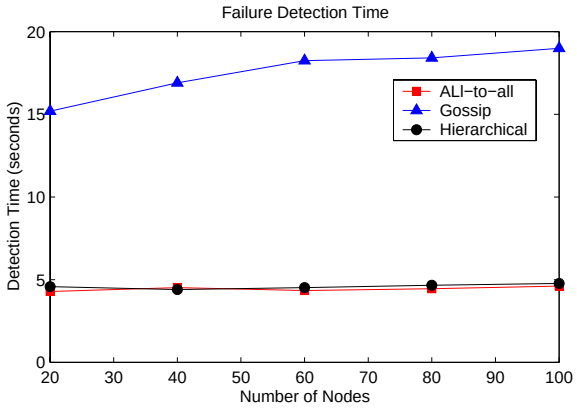


Figure 12: Failure detection time

## 6.5 View convergence time

Figure 13 compares the view convergence time. The hierarchical scheme has the similar view convergence time as the all-to-all scheme. This is because they have the same failure detection time and when a failure is detected, group leaders can quickly propagate this information to all nodes. Figure 13 also shows the view convergence time of the gossip scheme is the largest among the three schemes and it grows along with the number of nodes. Again, the hierarchical and the all-to-all schemes perform better than the gossip scheme.

## 6.6 Discussion

From the above experiments, we can see that gossip scheme performs the worst. Using the same amount of network traffic, it has the longest detection and convergence time. When the number of nodes scales up, the gossip scheme increases bandwidth usage quadratically. The reason is that each gossip has to carry a host's local view of the group membership, while the other two protocols use much smaller message size. If the

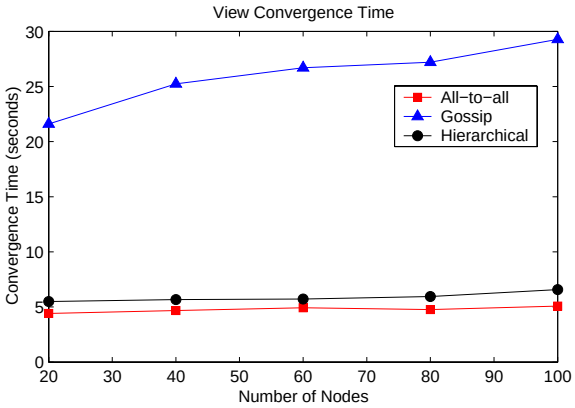


Figure 13: View convergence time

gossip protocol is organized into a hierarchical fashion [23, 9], the detection time and convergence time will be longer due to cross group gossips. However, it is shown that Gossip protocol is useful for large groups where each member only needs a partial view of group membership [9]. Gossip style approaches are also attractive when efficient multicast is not available such as wide-area applications. We focus on protocols, such as the all-to-all scheme and the hierarchical scheme that allow each member to quickly manage a global view. The hierarchical scheme is better than the all-to-all scheme for its less bandwidth consumption and comparable performance. In a word, our experiments have shown that the hierarchical approach is the most scalable among three approaches.

### 6.7 Effectiveness of Membership Proxy

In this experiment, we examine the effectiveness of the membership proxy protocol for a prototype search engine hosted in two data center, one of which is in the east coast and the other in the west coast. At second 20, the document retrieval service in the data center A fails. It recovers at second 40. Figure 14 shows the response time and throughput of the search engine service during the 60 seconds runtime. When the retrieval service fails in data center A, the overall throughput drops a little during the failure detection time. After that, the throughput matches the request arrival rate and the response time goes above 200ms. This is because the membership proxy is able to get the retrieval service from data center B with the cost of communicating through Internet instead of a high speed system area network. In our setting, the round trip time between two data centers is about 90 milliseconds. Thus, the service is still available during the failure with the help of the membership proxy protocol. When the retrieval service recovers in data center A, the response time quickly drops since all the requests are again serviced locally.

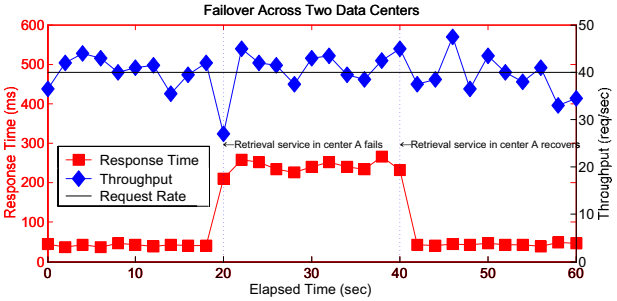


Figure 14: Effectiveness of membership proxy

## 7 Related Work

previous research on membership or failure detection protocols is rooted in multiprocessor machines, which requires precise and reliable membership services. Their studies emphasize on theoretical aspects of protocols [3, 6, 16, 7]. In a cluster environment, we can relax some of these requirements to provide a feasible and efficient membership protocol. This paper emphasizes on the design and implementation aspect of our hierarchical membership service.

Stok *et al.* described a hierarchical membership protocol which is most similar to ours. Compared to our work, the main difference is that their protocol requires all nodes have synchronized clocks so that the execution steps of the protocol can be synchronized. Additionally, there is no implementation of the protocol in the paper [22].

Gossip style membership services are different from heartbeat-based membership services in the aspect that they are based on probabilities [23, 13, 9]. This kind of protocols are more attractive in wide-area applications where full group membership is not required and multicast is not available. For instance, SCAMP [9] is a hierarchical variation of gossip protocols where group members only have partial knowledge of the group. We focus on the membership service for clustering middleware that provides full membership knowledge.

There are similar ideas of hierarchically organizing group members in the research of overlay networks [1]. However, most of the research is focused on wide area network applications, where network latency in these works is a concern and link bandwidth is heterogeneous. Failure detection time and view convergence time are not the primary goals.

There are many projects on High-Availability systems [14, 11]. Membership services are an essential component for these systems. Linux-HA provides a heartbeat based membership service [15]. It provides a hot-standby feature that one machine can take over another machine's IP when it is down. But it only works for small clusters. Our approach is proposed to manage large clusters and offload the failover feature to clustering middleware, which we think is more appropriate.

Resource monitoring tools, such as Ganglia [19] and the Network Weather Service [24] provide statistical information of machine resource or network resource of large scale clusters or computer grids to ease administration tasks. Our work

is to provide a complete view of the membership information to each cluster node to facilitate location transparent service invocation as well as load balancing.

The Metacomputing Directory Service in Globus project provides a static view of grid configuration [8]. Our membership service, on the other hand, provides dynamic information, such as CPU load and service status, which demands frequent updates. Furthermore, the membership directory is maintained on each node for efficient access which is required in a cluster environment.

# 8 Concluding Remarks

In this paper, we present a hierarchical membership service for large scale service clusters across multiple data centers. Inside each data center, we use a tree-based membership service that divides cluster nodes into multiple hierarchical groups. These groups are organized into a tree hierarchy to achieve good scalability. Across multiple data centers, we implement a membership proxy protocol to facilitate service invocation. We also provide a simple but effective interface to ease the administration task. Our evaluation shows the hierarchical membership service is scalable and efficient in large scale service clusters.

# References


[1] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable Application Layer Multicast. In *Proc. of ACM SIGCOMM'02*, Aug. 2002.

[2] E. A. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[3] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 322–330, New York, USA, 1996. ACM.

[4] R. Chow and T. Johnson. *Distributed Operating Systems and Algorithms*. Addison-Wesley, 1997.

[5] F. Cristian. Agreeing on processor group membership in timed asynchronous distributed systems. Technical Report CSE95-428, Dept. of Computer Science, UC San Diego, 1995.

[6] C. Fetzer. Enforcing perfect failure detection. In *21st Proceedings of the International Conference on Distributed Computing Systems (ICDCS2001)*, Phoenix, AZ, 2001.

[7] C. Fetzer and F. Cristian. A highly available local leader election service. *Software Engineering*, 25(5):603–618, 1999.

[8] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*, 1997.

[9] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2), February 2003.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *ACM SOSP*, 2003.

[11] Cluster Infrastructure for Linux. http://sourceforge.net/projects/ci-linux.

[12] I. Keidar. Moshe: A group membership service for WANs. In *MIT Technical Memorendum MIT-LCS-TM-593a, 1999.*, 1999.

[13] D. Kempe, J. M. Kleinberg, and A. J. Demers. Spatial gossip and resource location protocols. In *ACM Symposium on Theory of Computing*, pages 163–172, 2001.

[14] High-Availability Linux Project. http://www.linux-ha.org.

[15] Linux Heartbeat. http://www.linux-ha.org/heartbeat.

[16] G. Neiger. A new look at membership services. In *Proceedings of the fifteenth Annual ACM Symposium on Principles of Distributed Computing*, Philadelphia, Pennsylvania, United States, 1996.

[17] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, Mar. 2003.

[18] S. Raman and S. McCanne. A Model, Analysis, and Protocol Framework for Soft State-based Communication. In *Proc. of ACM SIGCOMM'99*, pages 15–25, Cambridge, Massachusetts, Sept. 1999.

[19] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler. Wide area cluster monitoring with ganglia. In *Proc. of the IEEE Cluster 2003 Conference*, Hong Kong.

[20] K. Shen, T. Yang, and L. Chu. Cluster Load Balancing for Fine-grain Network Services. In *Proc. of International Parallel & Distributed Processing Symposium*, Fort Lauderdale, FL, Apr. 2002.

[21] K. Shen, T. Yang, L. Chu, J. L. Holliday, D. A. Kuschner, and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, Mar. 2001.

[22] P. Stok, M. Claessen, and D. Alstein. A Hierarchical Membership Protocol for Synchronous Distributed Systems. In *1st Europen Dependable Computing Conference*, LNCS 852, pages 597–616. Springer-Verlag, Oct. 1994.

[23] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Style Failure Detection Service. In *Proc. IFIP Int'l Conf. Distributed Systems and Platforms and Open Distributed Processing (Middleware '98)*, pages 55–70, 1998.

[24] 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*, 1998.