A Distributed Service Management Infrastructure for Enterprise Data Centers Based on Peer-to-Peer Technology

Chunqiang Tang, Rong N. Chang, and Edward So

IBM Thomas J. Watson Research Center
{*ctang,rong,edwardso*}*@us.ibm.com*

Abstract

This paper presents a distributed service management infrastructure called BISE. *One distinguishing feature of BISE is its adoption of the Peer-to-Peer (P2P) model in support of realtime service management. BISE offers significant advantages over existing systems in scalability, resilience, and manageability. Current P2P algorithms are mainly developed for the file-sharing applications running on desktops, which have characteristics dramatically different from enterprise data centers. This difference led us to design our own P2P algorithms specifically optimized for enterprise environments. Based on these algorithms, we implemented a P2P substrate called BiseWeaver (25,000 lines of Java code) as the core of BISE. Our evaluation on a set of distributed machines shows that BiseWeaver is efficient and robust, and provides timely monitoring data in support of proactive SLA management.*

1. Introduction

The service sector has undergone a rapid expansion in the past several decades. In the United States, services account for approximately three quarters of GDP and eight out of ten jobs. This trend has driven the Information Technology (IT) industry to shift its focus from the sales of computer hardware and software toward providing value-added IT services. Another trend in the industry is that many organizations increasingly rely on web applications to deliver critical services to their customers and partners. These organizations typically want to focus on their core competency and avoid unnecessary risks caused by sophisticated IT technologies. IT service providers are therefore encouraged to host the web applications in their data centers at reduced cost and with improved service quality.

An IT service provider that offers comprehensive enterprise computing services may run hundreds of data centers, and a single data center may host up to thousands of applications running on thousands of machines. The sheer scale and heterogeneity of hardware and software pose grand challenges on how to manage these data centers. In this paper, we study the next generation service management infrastructure for enterprise data centers. Among many requirements for such an infrastructure, we consider the following most essential.

- **Scalable**. The infrastructure must be sufficiently scalable and can evolve as the service sector grows.
- **Resilient**. As faults are common in large systems, the infrastructure must continuously operate even in the face of failures.
- **Autonomic**. The infrastructure must be self-organizing, self-healing, and self-tuning. Otherwise, the management complexity of a large system would incur a high cost of ownership.
- **SLA centric**. As businesses revolve around service level agreement (SLA), it is important to both the provider and its customers that the infrastructure supports business-aligned proactive SLA management.

Providing infrastructure support for Internet-scale applications has been explored in middleware systems such as Ninja [12]. Most these systems are based on a traditional client-server model. We argue that a Peer-to-Peer (P2P) model has competitive advantages in meeting the requirements of a large-scale service management infrastructure.

In recent years, the P2P model has been widely popularized by applications with millions of users, e.g., the file-sharing tool KaZaA [8] and the Internet telephony program Skype [4]. In a P2P model, all active computing entities (or *nodes*) are treated equal, and each node functions as both a client and a server. Nodes organize themselves into an overlay network and collectively route traffics and process requests. There is neither a single point of failure, nor the bottlenecks associated with a centralized system.

All current Internet-scale P2P applications harness desktop resources scattered at network edges. We argue that, as a technology, the P2P model is not restricted to these end-user desktop environments. We believe that large-scale enterprise data centers are the true arena where the P2P model can reach its full potential while avoiding the legal disputes (e.g., copyright) that cripple the technology. Data centers, however, are dramatically different from desktop environments. For example, data centers are much more stable, and their management tasks are more sophisticated.

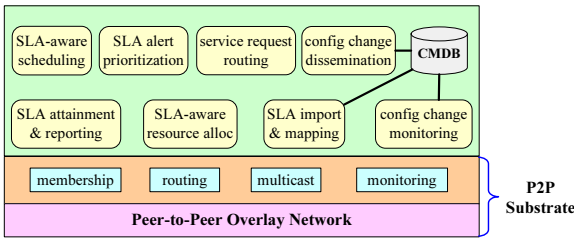


Figure 1. The architecture of our service management infrastructure *BISE*.

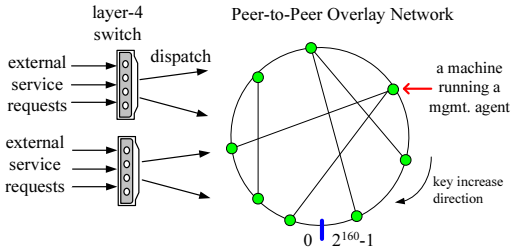


Figure 2. Our semi-structured P2P overlay network.

In our *Business-Aligned IT Service Environment (BISE)* project, we explore a novel P2P approach to managing machines and applications in data centers. The fundamental difference between desktop and data center environments in terms of stability leads us to design P2P algorithms that are dramatically different from those used in existing systems. We believe that we are among the first to implement a service management infrastructure based on P2P technology.

The remainder of the paper is organized as follows. Section 2 outlines the architecture of BISE. Section 3 presents the P2P algorithms specifically optimized for data centers. Section 4 describes the prototype implementation of BISE and the experimental results. Section 5 concludes the paper.

2. Architecture of BISE

Compared with existing systems, BISE has two distinguishing features. First, BISE is SLA centric and supports business-aligned proactive service quality management. Second, BISE employs novel P2P techniques to manage a large number of machines and applications. The high-level architecture of BISE is depicted in Figure 1. This design draws heavily from our past experiences with developing several key SLA management components [2, 3, 13]. These components, among other benefits, (1) enable an effective means of capturing and managing contractual SLA data as well as the provider's internal service management data, (2) provide on-demand evaluations of attained service levels, (3) prioritize SLA alerts based on business impact, and (4) automate the prioritization and execution of workflow processes based on a continual optimization technology. BISE extends these efforts from the traditional client-server model into a P2P architecture by incorporating a new P2P substrate called *BiseWeaver* as its core.

2.1. The BiseWeaver P2P Substrate

Unlike traditional service management systems, BISE is based on the P2P model and does not need dedicated management machines to monitor machines that run user applications. Instead, each machine hosts some user applications and also devotes some resources to collectively form the management infrastructure. BISE runs a management agent (a Java program) on each machine (in addition to the user applications running on those machines). Each agent connects to the agents on several other machines. Together, the agents form a self-organizing overlay network to carry out management tasks (see Figure 2).

In a legacy system, the BISE agent may not be able to run on every machine. Alternatively, we can run the agent on a subset of machines, and ask each agent to monitor and control some other machines through their legacy interfaces. This requires no fundamental changes to our architecture. For brevity, in the rest of this paper we assume that each machine runs a BISE agent, and use the terms "agent", "node", and "machine" interchangeably when the context is clear.

BiseWeaver, the P2P substrate in Figure 1, is the core of the service management infrastructure. It provides basic functionalities such as monitoring the availability of nodes, routing messages at the application level, multicasting a message to all nodes, and aggregating monitoring data from all nodes. BiseWeaver is self-organizing and fault-tolerant. It adapts as nodes come and go, and does not rely on any special nodes to function,

2.2. Multicast for Data Dissemination

Multicast is a communication paradigm in which a node efficiently sends a message to multiple recipients [5]. Because reliable *IP-level* multicast is typically not supported by the underlying network, BISE implements its own reliable *application-level* multicast. In the simplest form, a node multicasts a message by sending the message to each of its neighbors in the overlay. Unless a receiver of the message has seen the message before, it further forwards the messages to each of its neighbors, except the neighbor from which the message arrived. With this simple mechanism, a multicast message quickly floods through every link in the overlay, and may reach a node through different paths. Section 3 will present a more efficient multicast implementation based on tree and gossip.

Multicast can be used to disseminate system configurations. In one scenario, an administrator changes some configuration (e.g., introducing a new service policy) and stores the change into the Configuration Management Database (CMDB). The infrastructure then automatically uses multicast to deliver this change to all machines. In another scenario, when a machine starts a new user application, it mul-

ticasts this information to others so that they know where to forward requests for this application.

2.3. Monitoring Data Aggregation

BISE builds a tree embedded in the overlay network to support efficient monitoring (and metering) data aggregation and scalable realtime SLA management. The tree includes all nodes. It is constructed in a decentralized fashion and adapts as nodes come and go. Each internal node of the tree collects monitoring data from its children in the tree and sends the aggregated data to its parent. Eventually, the aggregated monitoring data of the entire system arrive at the root node of the tree. Further detail of this tree protocol is presented in Section 3.

The aggregated monitoring data include anomaly events at both the infrastructure level and the application level, e.g., the failure of one machine's disk or repeated timeouts when connecting to a database. By integrating operational data on SLA contracts, failure notifications, and dependencies between system components and active business transactions, BISE can generate and prioritize SLA alerts for potential SLA breaches. This advanced capability is not supported by most existing SLA management systems.

The aggregated monitoring data also include information that facilitates resource allocation and provisioning, e.g., the request rate for each application and the capacity and utilization of resources such as CPU, memory, disk, and network. We have developed an algorithm that takes this information as input and automatically determines the applications to run on each machine to optimize certain objectives.

2.4. Configuration Management Database

The Configuration Management Database (CMDB) is the central place that tracks the configuration of the entire system. Historical data show that a significant fraction of outages are caused by improper executions of changes to system configurations. When information in CMDB is incomplete or inaccurate, a misunderstanding can cause a chain reaction that leads to cascaded failures.

Our previous discussion assumes that the configuration changes are properly entered into CMDB, and then propagated throughout the system using multicast. In reality, changes may not be properly recorded in CMDB. BISE facilitates automatic discovery of discrepancies between real configurations and information in CMDB. Each agent monitors the configuration of its local machine and reports unexpected changes as part of the monitoring data. When the aggregated data arrive at the root node of the tree, the root can record the unexpected changes into CMDB and send an alert to the administrator.

CMDB may also be modified by a component that imports SLA contracts and maps them to internal SLA management data. The issues involved are non-trivial due to the diversity of SLA contracts and the mismatch between external contract presentation and the system's internal SLA management data. We studied many real SLA contracts and developed a flexible language that captures common SLA terms and supports extensions and plug-ins [13].

2.5. Service Request Routing

Traditionally, data centers use dedicated front-end nodes as load balancers to direct external requests to back-end servers. As the system scales, these front-end nodes may become bottlenecks and require careful provisioning. Under the P2P model, each node acts both as a back end to process requests that it can handle, and as a front end to forward to others requests that it cannot process locally. The difference between front-end and back-end nodes disappears.

With this P2P model, external requests entering a data center are evenly distributed to all nodes using one or more layer-4 switches (see Figure 2). A layer-4 switch only understands the protocols up to the TCP layer. This simplicity allows efficient hardware implementation and ensures that they are not bottlenecks. When a node receives a request from a layer-4 switch, it knows where to forward the request because, from the received configuration data, it knows nodes that can handle the request. A request may be forwarded through multiple hops for load balancing. Our BISE prototype includes an implementation of this request routing mechanism.

3. The Peer-to-Peer Substrate

BISE's distinguishing features include a focus on business SLA and its adoption of the P2P model. Due to space limitations, we present in this paper the detail of the P2P substrate (BiseWeaver), and refer readers to other papers [2, 3, 13] for the detail of other components.

BiseWeaver organizes all nodes into an overlay network and builds a tree embedded in the overlay. Neighbors in the overlay monitor one another's health. The tree is used to disseminate system configurations and to aggregate monitoring data. The fundamental difference between desktop and data center environments in terms of stability leads us to design P2P algorithms that are dramatically different from those used in existing P2P file-sharing application. Below, we first discuss the difference between desktop and data center environments, and then present our P2P protocol.

3.1. Desktop vs. Data Center

The design of most existing P2P systems assumes that nodes are very unstable. For example, when a KaZaA [8] user wants a song, she starts a program to join an overlay, downloads the song, and then closes the program. The whole process takes only a few minutes [7]. Because of this short lifetime of nodes in the overlay, a node knows only their direct neighbors and does not keep state information about remote (non-neighbor) nodes. The rationale is that even if a node X knows a remote node Y, it is likely that before X ever needs to contact Y for any reason (e.g., to forward a service request), Y already left the overlay.

This reasoning, however, is not valid for data center environments, where machines are stable and service request rate is high. In such an environment, if a node X gathers states about remote nodes, X can use this information to directly forward many requests to those remote nodes before they leave the overlay. Otherwise, node X would have to resolve those requests through inefficient multi-hop routing. In data center environments, the network traffic created by more aggressively maintaining node states is well compensated for by savings from efficient routing.

Our previous analysis [10] formally gives the optimal number d of other nodes about which a node should know in order to minimize the total network traffic. Here we do a first principal calculation to show that it is actually feasible for a node to know all other nodes in an overlay. Suppose a data center has 2,000 machines and each machine reboots every 72 hours. (In reality, machines in data centers reboot much less frequently.) On average, the whole system has one machine reboot every 130 seconds. During a reboot, the agent running on the machine first leaves and then rejoins the overlay. Using either IP multicast or application-level multicast, this membership change can be easily notified to all nodes without exhausting network resources, because membership messages are small (around 50 bytes) and membership changes are infrequent (once every 130 seconds).

3.2. Protocol of the Overlay

BiseWeaver builds an overlay network that is different from existing structured or unstructured overlay networks. It is *semi-structured*, as it only maintains a minimum ring structure (see Figure 2). The ring corresponds to key range [0, 2^{160}-1]. Each node is assigned an identifier in this key range and its position on the ring is determined by its identifier. A node's identifier is the SHA-1 hashing of its IP address.

Like BiseWeaver, existing structured overlays such as Chord [9] and Pastry [6] also use a ring, but they construct additional structures. Chord builds a small-world graph and Pastry builds a hypercube. By contrast, BiseWeaver is much simpler. Its only strict structure is the ring; all other links are chosen at random without any topological restrictions. This simplicity makes BiseWeaver easy to build and maintain. A node keeps a constant (e.g., $k = 4$) number of random neighbors in addition to its two neighbors on the ring.

A node in BiseWeaver maintains a membership table that records all live nodes in the system, even if it never directly communicates with some of them. When a node joins or leaves, this membership change is broadcast to all nodes. At any given point in time, a node's membership table may be incorrect: it may include some dead nodes or omit some live nodes. However, these errors are transient, and will not prevent the overlay from proper function. The knowledge of full membership is merely an optimization. Our protocol ensures that eventually all membership tables of nodes converge to a consistent state. Moreover, because data center environments are relatively stable, the membership tables are accurate most of the time.

3.2.1. Node Join

We give some definitions before introducing the protocol. Given a set $\mathcal{S}$ of n nodes, we sort them in increasing order by identifier. Let $(s_0, s_1, \cdots, s_{n-1})$ denote this sorted list. The successor of node $s_j, 0 \leq j < n$, in set $\mathcal{S}$ is s_{j+1}. The predecessor of $s_j, 0 < j \leq n$, in set $\mathcal{S}$ is s_{j-1}. The successor of s_{n-1} in set $\mathcal{S}$ is s_0. The predecessor of s_0 in set $\mathcal{S}$ is s_{n-1}. Note that a node's predecessor (or successor) is different in different sets.

When a new node N joins, we assume that it knows (through some out-of-band method) about at least one node B already in the overlay. Node N sends a bootstrap request to node B. Node B replies with node N's predecessor P among nodes in B's membership table. Node N then sends a request to its predecessor P to join the ring. Node N also copies a complete membership table from node P so that N knows all other nodes in the system.

The event that node N just joined is broadcast to all nodes. Node P first sends a message to notify each of its neighbors. Unless a receiver of the message has seen the message before, it further forwards the messages to each of its own neighbors, except the neighbor from which the message arrived. Quickly the message floods through every link in the overlay. A node with m neighbors may receive the message up to m times. Like the use of message redundancy in gossip protocols [1], the redundancy in membership notifications ensures the reliability of the messages. Because membership changes are infrequent and membership messages are small (around 50 bytes), they consume little network bandwidth.

In addition to joining the ring, the new node N also tries to establish links to k random nodes (typically $k = 4$). It randomly chooses k nodes from its membership table and sends a request to each of them. A receiver Y accepts this request if Y does not have an excessive number of neighbors, e.g., $rand\_nbr(Y) \leq k + 1$, where $rand\_nbr(Y)$ is the number of Y's random neighbors.

On average, each node has k random neighbors, one predecessor along the ring, and one successor along the ring. As nodes come and go, the number of neighbors of a node may change. A node X periodically adds or drops links to maintain a proper number of neighbors. If $rand\_nbr(X) \leq k - 2$, node X adds one random neighbor. If $rand\_nbr(X) \geq k + 2$, node X drops one random neighbor; it prefers to drop random neighbors that have a large number of neighbors. A node never voluntarily drops its predecessor or successor.

3.2.2. Maintenance Operations

Each node periodically exchanges heartbeats and probes with other nodes to maintain the integrity of the ring and the accuracy of its membership table.

Heartbeats. Heartbeat messages help maintain the integrity of the ring. Each node Y periodically sends a message to its successor Z, stating that Y is Z's predecessor. Likewise, each node periodically sends a message to its predecessor. If node Y fails, its successor Z detects this though the loss of heartbeats from Y. Node Z deletes Y from its membership table and broadcasts a message to announce the death of Y. Node Z then tries to find a new predecessor. Node Z sends a request to Z's predecessor X among nodes in Z's membership table. If node X is also dead, node Z times out, deletes X from its membership table, and announces the death of X, and so forth. This procedure ensures that, if the ring is broken into fragments, it will eventually be reconnected.

The ring is correct if every node has a correct successor. However, errors may occur as nodes come and go. Suppose node X is the correct predecessor of node Y, which in turn is the correct predecessor of node Z. If node X somehow is unaware of the existence of node Y, both nodes X and Y would consider node Z to be their successor. Upon receiving heartbeats from nodes X and Y, node Z can discover this error and informs X that Y should be X's successor.

Probes. Probe messages help maintain the accuracy of nodes' membership tables. Each node N periodically picks a node X from its membership table and sends a probe message to X. Upon receiving the probe, node X does two things. First, if node N is unknown to node X, X adds N to its membership table and broadcast a message to announce N's existence. Second, among nodes in X's membership table, X identifies X's successor S_X and N's successor S_N. Node X sends the ID of S_X and S_N to node N. Upon receiving the response, node N inserts S_X and S_N into its membership table unless they are already there.

If node N times out while waiting for a response for its probe to node X, N retries up to five times. If all retries fail, N delete X from its membership table and broadcasts a message to announce the death of X.

Suppose the probing period is T (typically T=30 seconds). If node N probes node X in cycle j, N probes X's successor (among nodes in N's membership table) in cycle j+1. That is, a node follows the ring to probe other nodes. Eventually, a node will probe all nodes in its membership table, detecting all dead nodes and discovering all live nodes. The first node that a node probes is chosen at random and nodes start their probing asynchronously. This randomness ensures that any given node is likely to be probed by some other nodes during a short period time. Therefore, dead nodes are discovered quickly. Specifically, it can be proven that the probability that no nodes probe a given node during h consecutive probing cycles is e^{-h}, where $e \approx 2.7$ is the natural number.

3.3. Correctness of BiseWeaver

An overlay is partitioned if no overlay path exists between some nodes. Overlays such as Chord and Pastry have strong connectivity, but cannot recover once partitioned. By contrast, BiseWeaver uses extra information in membership tables to recover from partitioning and ensure that the ring and membership tables are eventually consistent.

We first give some definitions that will be used in the proof of the correctness of our protocol. The membership tables of nodes in an overlay induce a *membership graph*, in which a vertex corresponds to a node and there is an edge between two nodes X and Y if either X is in Y's membership table, or Y is in X's membership table, or both. An overlay is *membership connected* if its membership graph has at least one path between any two nodes.

Note that even if BiseWeaver is partitioned, it is still very likely to be *membership connected*, because each membership table includes almost all nodes in the system. Also note that BiseWeaver can be *membership connected* even if the membership tables of many nodes are inaccurate.

Theorem 1. *If BiseWeaver is "membership connected", given sufficient time after the last node joins or leaves, each node will find its correct successor, and each node's membership table will include all and only live nodes. That is, both the ring and the membership tables will be consistent.*

Proof. Due to space limitations, we only give the highlights of the proof. The basic idea is that, after dead nodes are excluded from the membership tables through the membership probing process, some live nodes start to form a ring that includes a subset of live nodes in the system. Because the overlay is membership connected, nodes outside the ring talk to nodes on the ring and are gradually recruited into the ring. Eventually, the ring includes all live nodes in the system. (It is also possible that the live nodes initially form multiple disjoint rings. If so, each ring grows independently and eventually they merge into a single ring.) Because a node follows the ring to probe other nodes, eventually its membership table includes all live nodes. Because a node probes every node in its membership table, eventually all dead nodes are purged from its membership table. Therefore, both the ring and the membership tables are consistent. □

3.4. Advantages of BiseWeaver

The correctness proof of our protocol uses the fact that a node sequentially walks through the ring to detect dead nodes and to discover live nodes. In reality, this happens in parallel. Each node probes a different section of the ring. When it discovers a dead node or a missing live node, it broadcasts this information to all other nodes. Working together, nodes quickly detect dead nodes and discover missing live nodes.

Although a membership table records all nodes in the system, the maintenance overhead is actually very low. Because machines in data centers are very stable, changes to membership tables are infrequent. In our default configuration, every 10 seconds, a node sends a heartbeat message

to its predecessor and successor, respectively; and every 30 seconds, a node sends a probe to a random node. The storage overhead of the membership table is also low. Assuming that it costs 128 bytes to store the basic information regarding a node (e.g., IP and port number), a membership table with information for 1,000 nodes is only about 128KB.

BiseWeaver is *semi-structured*. It has three major components: a ring, a random overlay, and the replicated membership tables. This unique design offers several advantages in data center environments. (1) It is simpler than structured overlays such as Chord or Pastry. BiseWeaver's only mandatory structure is the ring. (2) Like Chord and Pastry, BiseWeaver's simple ring structure can support the functionality of a distributed hash table (DHT), which is not supported by unstructured overlays. (3) Owing to the complete membership tables, BiseWeaver can directly deliver messages between the source and the destination, as opposed to going through inefficient multi-hop routing as in Chord and Pastry. BiseWeaver introduces less total traffic because of this routing efficiency. (4) Unlike Chord and Pastry, BiseWeaver can repair itself and evolve into a consistent state even if the overlay is partitioned.

3.5. Protocol of the Tree

BiseWeaver uses a tree to aggregate monitoring data from all nodes. It selects overlay links in a decentralized fashion to form a tree embedded in the overlay. The algorithm to build the tree is similar in spirit to the classical Distance Vector Multicast Routing Protocol (DVMRP) [5], but note that we only need a single tree. The tree conceptually has a root and the tree links are overlay links on the shortest paths (in terms of latency) between the root and all other nodes. The root node is the one with the smallest identifier among all nodes. If the root fails, its successor announce its death to all nodes and the next live node with the smallest identifier automatically becomes the root. A root periodically broadcasts a "root message" to all nodes, asserting that it is the root. If multiple nodes consider themselves as the root, which may be caused by errors in the membership tables, they discover each other through the "root messages", and the true root wins.

Our discussion so far assumes that a multicast message floods through every link in the overlay, which is reliable but wastes network resources if the multicast message is large. A more efficient method only forwards a multicast message along the tree links so that each node receives the message only once. To ensure the reliability of multicast messages, the overlay neighbors exchange gossips with one another to discover and recover lost multicast messages. See our GoCast protocol for detail [11].

The root node of the tree periodically gathers monitoring data from all nodes. At the beginning of a gathering cycle, the root reliably multicasts a request to all nodes. After receiving the request, a leaf node of the tree immediately sends its monitoring data to its parent in the tree. An internal node of the tree aggregates its own data and data received from its children, and then sends them to its parent. An internal node may apply plug-ins to reduce the size of the aggregated data. If an internal node P has a child C that is slow in sending C's data to P, P will not faithfully wait for C, because P is uncertain if C is still alive. Instead, after receiving the request from the root, node P waits for at most $h = 0.5$ seconds before it sends the data received so far to its parent. If node P receives data from all of its children before waiting for h seconds, it sends the data to its parent immediately. Node P forwards to its parent data that arrive late (after h seconds) in a pipeline fashion. Periodically every $g = 0.1$ seconds, it sends recently arrived data, if any, to its parent. This process ensures that the root gets data quickly while the message rate in the system is moderate.

4. Experimental Results

The design of BISE draws heavily from our past experiences on SLA management and peer-to-peer systems. In the past, we have developed and evaluated several key SLA management components in a traditional client-server environment [2, 3, 13]. BISE extends these efforts into a P2P architecture. We designed and implemented a new P2P substrate (BiseWeaver) as the core of BISE. As the SLA management components have been documented before, the evaluation in this paper focuses on BiseWeaver.

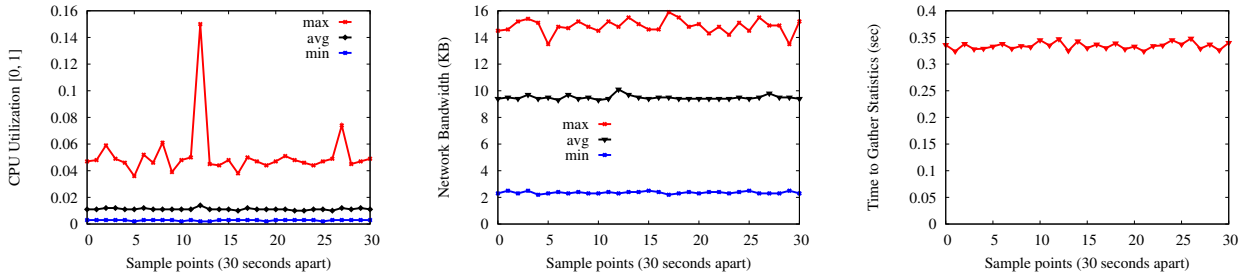
BiseWeaver implements the protocols described in Section 3, including overlay construction and maintenance, tree construction and maintenance, configuration data dissemination, monitoring data aggregation, and P2P-style service request routing. Our current implementation BiseWeaver consists of 25,000 lines of Java code.

BiseWeaver boots from a single machine. Taking as input the JAR file of the agent program and a list of machines in a data center, a booting script uses the secure file transfer tool SCP to copy the JAR file to each machine and uses the secure login tool SSH to start the agent on each machine. The agent on one machine boots first and serves as the bootstrapping node to help other agents to join the overlay.

Ideally, we would like to experiment with thousands of machines. Due to limited resources in the development phase, we experimented with systems consisting of up to 3,000 agents running on 51 machines with a total of 164 processors. Agents on the same machine use different network ports and function independently. The experiments presented in this paper run about 1,000 agents on 37 machines with a total of 87 processors. In the rest of this section, we refer to an agent as a node in the overlay.

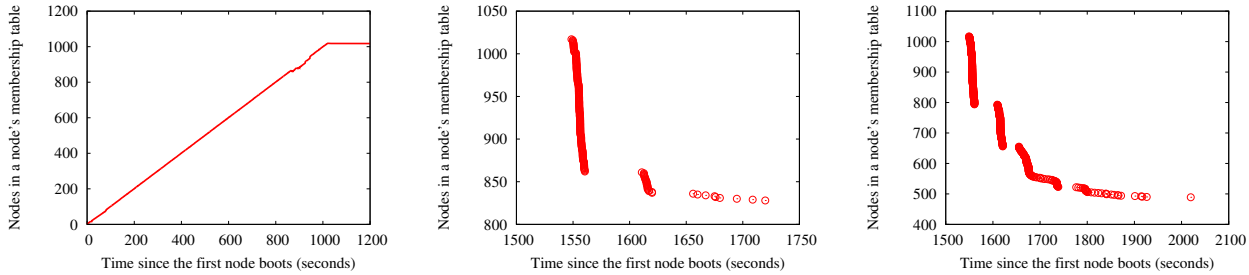
4.1. Scalable and Efficient Monitoring

BiseWeaver supplies the SLA management components with timely monitoring data of the entire system. Every T=30 seconds, BiseWeaver aggregates the monitoring data through a tree embedded in the overlay and store the aggregated data at the root of the tree. The aggregation period T



(a) CPU. (b) Network. (c) Time to collect data.

Figure 3. Collecting realtime monitoring data.



(a) Booting process. (b) 20% of nodes fail. (c) 50% of nodes fail.

Figure 4. The number of nodes in a node's membership table as nodes come and go.

is adjustable. An SLA management component that is interested in this data can query any node in the overlay to obtain the IP address of the root, and then register with the root to fetch this data periodically.

Figure 3 presents examples of the aggregated monitoring data. Figure 3(a) shows the minimum, average, and maximum CPU utilization of machines, after the 1,020-node overlay grows to its full size. The data are sampled throughout the system every 30 seconds. Machines have different levels of CPU utilization, because machines are of different speed and host different applications. In this experiment, about 1% of the CPUs are utilized on average, showing that BiseWeaver imposes a very low overhead, considering that each processor runs about 12 agents and the system frequently aggregates monitoring data from all nodes (once every 30 seconds).

Figure 3(b) shows the network bandwidth consumption of machines. The data are collected from Linux's "/proc/net/dev" interface, which counts all data communications on a machine. BiseWeaver only contributes to part of this bandwidth consumption. The average bandwidth consumption is very low even if we collect monitoring data from 1,020 nodes every 30 seconds. Because we run multiple agents on each machine, a production system that runs only one P2P agent on each machine should experience one order of magnitude less bandwidth consumption than the numbers in this figure. The bandwidth consumption can be further reduced if we collect monitoring data less frequently, e.g., every 5 minutes as opposed to every 30 seconds.

BiseWeaver collects monitoring data from 1,020 nodes every 30 seconds. Figure 3(c) shows the delay between the time that the root starts to multicast a data-gathering request and the time that the data from the last node arrive at the root. This figure demonstrates that BiseWeaver can gather monitoring data rapidly. The whole process took less than 0.4 seconds every time. This speed is mainly due to the P2P architecture, in which nodes work collaboratively to gather monitoring data.

4.2. Membership Table Maintenance

Next, we evaluate the freshness of information in nodes' membership tables as nodes come and go. Figure 4(a) shows the number of nodes in a node's membership table when the overlay starts from a single node. In this experiment, we arrange one new node to join the overlay every second until it grows to its full size (1,020 nodes). This figure shows that the membership table is updated quickly and is kept accurate even during the rapid growth period,

The next experiment introduces node failures into the system. The overlay grows to its full size at around 1,020 seconds since the first node boots. At around time 1,500 seconds, the root of the tree multicasts a message to all nodes. Upon receiving this message, a node immediately fails with a probability of 0.2. Therefore, about 20% of the 1,020 nodes in the overlay fail almost *concurrently*. Figure 4(b) shows the number of nodes in a live node's membership table after this massive failure happens. With our current configuration, a live node N sends a heartbeat to its successor

P every 10 seconds. If node P does not receive any heartbeats from node N for 50 seconds, P considers N dead and broadcasts a notification throughout the system.

In Figure 4(b), the first wave of dead node eviction starts at time 1,550 seconds. Within just 10 seconds, the number of nodes in the membership table quickly drops from 1,020 to 862. The second wave happens 50 seconds later between time 1,610 seconds and 1,619 seconds; the number of nodes in the membership table further drops from 862 to 837. The third wave starts at time 1,656 seconds; a total of 10 dead nodes are sporadically detected over a period of 63 seconds.

The dead nodes evicted in the first and the second waves are detected by their successors on the ring. The 10 dead nodes evicted in the third wave is mainly discovered by probing messages exchanged between random nodes. In our current configuration, a node N selects a random node P to probe every 30 seconds. If node N receives no response, it retries up to 5 times in 150 seconds and then announces the death of node P. This is the reason why the third wave starts at around time 1,500+150=1,650 seconds. In summary, when 20% of the 1,020 nodes fail concurrently, BiseWeaver repairs itself and returns to a consistent state in less than 4 minutes.

The experiment in Figure 4(c) is the same as that in Figure 4(b), except that, in Figure 4(c), 50% of the nodes fail concurrently. Because of this massive failure, the ring is broken into fragments and the overlay is partitioned. (For the sake of efficiency, a node in BiseWeaver maintains only about six neighbors. If nodes have more neighbors, the partitioning may be avoided.) Under this extreme scenario, 94% of the dead nodes are evicted from the membership table within 4 minutes, and the membership table becomes completely accurate within 9 minutes. Note that the overlay and the ring are repaired much earlier before the membership tables become accurate.

Maintaining a full membership table on every node is one of BiseWeaver's most prominent features. Overall, the results in Figures 4(a), (b), and (c) suggest that BiseWeaver updates the membership tables in a timely fashion even in the face of rapid node joins or massive node failures. The massive failures in Figures 4(b) and (c) are unlikely to happen in data centers. Even if they do happen, BiseWeaver can repair the ring and the overlay quickly. It is slower to clean up the membership tables, but the whole process still completes within a few minutes. We believe that, in large-scale data centers, our protocol can efficiently maintain accurate membership tables on all nodes.

5. Conclusions

This paper presented a service management infrastructure called BISE. One distinguishing feature of BISE is its adoption of the P2P model in support of realtime service managements. Compared with existing systems, BISE offers significant advantages in scalability, resilience, and manageability. We believe that we are among the first to apply the P2P model to large-scale service management. We take the P2P technology beyond simple file sharing into sophisticated service management in enterprise data centers. We are also among the first to articulate the difference between desktop and enterprise data center environments, and propose new P2P algorithms specifically optimized for data centers. The BISE P2P substrate, BiseWeaver, is semi-structured. It has three major components: a ring, a random overlay, and the replicated membership tables. This unique design offers several advantages over state-of-the-art structured overlays. Our main future work is to enhance BISE for a production environment.

Acknowledgments

We thank Michael Frissora and James V. Norris for their enormous efforts in maintaining the experimental platform. We are grateful to Fausto Bernardini and Manoj Kumar for their management support.

References

[1] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999.

[2] M. J. Buco, R. N. Chang, L. Z. Luan, E. So, C. Tang, and C. Ward. PEM: A Framework Enabling Continual Optimization of Workflow Process Executions Based upon Business Value Metrics. In *SCC*, 2005.

[3] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu. Utility computing SLA management based upon business objectives. *IBM Systems Journal*, 43(1), 2004.

[4] KaZaA. http://www.kazaa.com.

[5] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, 2000.

[6] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[7] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *MMCN*, San Jose, CA, USA, 2002.

[8] Skype. http://www.skype.com.

[9] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[10] C. Tang, M. J. Buco, R. N. Chang, S. Dwarkadas, L. Z. Luan, E. So, and C. Ward. Low Traffic Overlay Networks with Large Routing Tables. In *ACM SIGMETRICS*, 2005.

[11] C. Tang, R. N. Chang, and C. Ward. GoCast: Gossip-enhanced Overlay Multicast for Fast and Dependable Group Communication. In *DSN*, 2005.

[12] J. R. von Behren, E. A. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A framework for network services. In *USENIX Annual Conference*, 2002.

[13] C. Ward, M. J. Buco, R. N. Chang, L. Z. Luan, E. So, and C. Tang. Fresco: A Web Services based Framework for Configuring Extensible SLA Management Systems. In *ICWS*, 2005.