RC24721 (W0901-037) January 12, 2009
Computer Science

IBM Research Report

Blue Eyes: Scalable and Reliable System Management for Cloud Computing

Sukhyun Song
Department of Computer Science
University of Maryland
College Park, MD

Kyung Dong Ryu, Dilma Da Silva
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Blue Eyes: Scalable and Reliable System Management for Cloud Computing

Sukhyun Song[1], Kyung Dong Ryu[2], Dilma Da Silva[2]

[1]*Department of Computer Science*
University of Maryland, College Park, MD
shsong@cs.umd.edu

[2]*IBM T.J. Watson Research Center*
Yorktown Heights, NY
{kryu, dilmasilva}@us.ibm.com

Abstract

With the advent of cloud computing, massive and automated system management has become more important for successful and economical operation of computing resources. However, traditional monolithic system management solutions are designed to scale to only hundreds or thousands of systems at most. In this paper, we present Blue Eyes, a new system management solution with a multi-server scale-out architecture to handle hundreds of thousands of systems. Blue Eyes enables highly scalable and reliable system management by running many management servers in a distributed manner to collaboratively work on management tasks. In particular, we structure the management servers into a hierarchical tree to achieve scalability and management information is replicated into secondary servers to provide reliability and high availability. In addition, Blue Eyes is designed to extend the existing single server implementation without significantly restructuring the code base. Several experimental results with the Blue Eyes prototype have demonstrated that our multi-server system can reliably handle typical management tasks for a large scale of endpoints with dynamic load-balancing across the servers, near linear performance gain with server additions, and an acceptable network overhead.

1. Introduction

Today, there are more than a hundred million computing devices connected to the Internet. These networked devices include high-end servers, workstation PCs, network routers, and printers. A significant portion of them are owned by corporate and large organizations, and managed in groups through system management tools in a semi-automated way. Massive and automated management of these networked computing resources becomes even more critical with introduction of a new cloud computing paradigm where numerous workloads are expected to migrate from a small number of dedicated machines in separate domains to a securely-shared and virtualized computing pool in mega-scale datacenters. In addition, efficient and effective management of computing systems can dramatically reduce overall IT costs as recent data shows that system management costs outweigh acquisition costs.

However, traditional system management solutions that are in use in the field are targeted at the scale of only hundreds or thousands of systems at most. The inherent scalability limitation of such solutions is due to its single server architecture. Simple tasks such as discovering systems on the network and collecting inventory can easily overwhelm a high-end server when its target endpoints are in the size of thousands or more. Its only option to scalability is a *scale-up* approach: replacing the management server with a more powerful machine with more processors and memory.

We believe a true solution to this scalability bottleneck is the multi-server *scale-out* architecture, where management servers can be added on demand to increase management power when the target system to be managed grows. Similar approaches have been attempted. However, their requirement of significant changes into the legacy software kept them from being adopted. In addition, there is an increasing need for aggregating multiple existing management domains into one due to mergers and acquisitions. The multi-server architecture can provide a seamless integration of system management domains by simply grouping management servers into one management server network. A server can continue to operate for its own domains as it did before while providing its local information and accessibility to its endpoints to other servers when requested.

Another requirement for system management for a large datacenter is its reliability and availability. Downtime of management servers can mean delayed deployment of critical security patches and untimely reconfiguration of cloud resources to respond to rapidly changing computation demands, both of which are costly for production systems.

In this paper, we introduce the Blue Eyes system management architecture and its prototype that provide

high scalability and reliability using a structured network of management servers. Blue Eyes is designed to provide the following five important characteristics.

- **Dynamic scaling**: management servers can be added dynamically in a scale-out manner to meet the growing demand for management capacity.
- **Domain merging**: management servers can be merged into one management domain when administration sub-domains are combined.
- **Distribution transparency with locality**: users can connect to any server in the server network to manage any endpoint while endpoint locality can be exploited when requested.
- **Failure Resistance**: all the endpoints are still manageable even in the presence of a management server failure.
- **Easy applicability to existing solutions**: Changes to the existing legacy management software are mostly limited to the addition of server-server interactions, not affecting implementation of core management functions between the server and managed endpoints.

This paper is organized as follows. Section 2 describes the design and core algorithms of Blue Eyes that provide scale-out scalability and reliability. Section 3 presents core management operations and tasks that Blue Eyes provides and performs. Section 4 describes the current implementation of the prototype. Section 5 evaluates the Blue Eyes prototype by demonstrating how it achieves its scalability and reliability goals. Section 6 lists the related work and contrasts it to our work. Section 7, finally, concludes the paper with summary and future work.

2. Design

In this section, we describe the overall design and architecture of Blue Eyes and how it can satisfy both scalability and reliability requirements in large-scale system management required in cloud computing.

2.1 Overall Architecture

Unlike the traditional system management solutions (Figure 1), our Blue Eyes system is designed to maintain multiple management servers (Figure 2) by scaling out from a single management server. In a nutshell, the systems and networked devices to be managed (referred to as *managed endpoints*) are partitioned to be assigned to multiple management servers so that management workload can be balanced. For example, in Figure 2, all the servers collaboratively manage different groups of endpoints using a management server network while the traditional system management server in Figure 1 has to directly manage all the endpoints.

In addition to dynamic scalability, a system management solution also requires server failure resistance to provide reliability of system management. This becomes more important and also viable as we adopt the multi-server architecture. To this end, Blue Eyes maintains another structured network of backup servers and provides a *detect-and-switch* type of fail-over mechanism. A simple example is shown in Figure 2 where one primary server and one secondary server are collocated on each machine. The primary server *A* has the corresponding secondary server *A'* which is placed on a different machine.

The structure of and the interactions among management servers to provide dynamic scalability and reliability in Blue Eyes are described in the following subsections in detail.

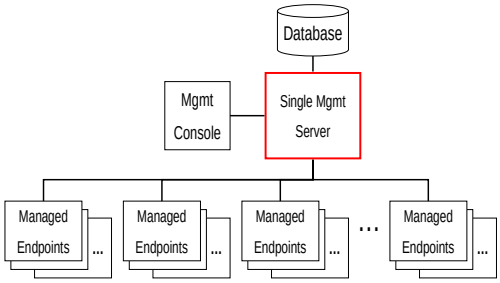

Figure 1: Traditional System Management Architecture

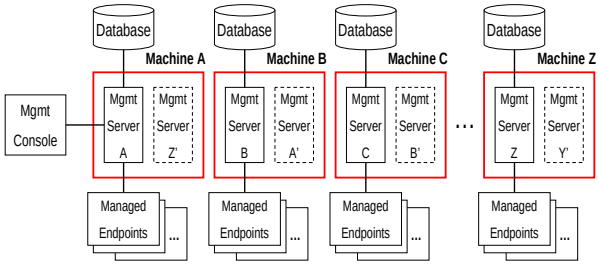

Figure 2: Blue Eyes Multi-Server System Management Architecture

2.2 Management Server Tree

In Blue Eyes, a group of management servers construct a management network for its communication for collaborative operations. We chose a hierarchical tree topology for two reasons. First, global server information and system monitoring data can be naturally aggregated upward when it is collected and central information can be easily propagated down the tree to all the servers. Second,

system administration domains and network domains are often structured following the business organization hierarchy. In this case, management task locality can be exploited by using only the relevant sub-tree for domain-specific management operations. The alternative centralized star topology with one master server and all slave servers, despite its simplicity, causes a bottleneck problem. On the other hand, the use of DHT for P2P structuring among management servers would suffer from its unbeneficial complexity as management server additions and deletions do not occur frequently. In our prototype, we use a binary tree, which is referred to as *Management Server Tree (MST)*.

An MST is implicitly constructed using the tree index numbers of the servers. This tree index information is shared through a globally shared data structure called *Management Server List (MSL)*. An example of the server tree is depicted in Figure 3. Each node is a management server denoted by its tree index number. A management server with the tree index number 1 becomes the root node in the tree. Since all management servers share the MST structure information, a user's console can connect to any server to access and manage any endpoints. When the MST changes by adding or deleting a management server, the tree index number for each server can be changed accordingly, and the updates are forwarded downward through the new tree structure.

The MST is exploited to balance the communication workload among servers. Suppose we need to execute a management task for all the endpoints in the entire management domain. The management task should be forwarded to all the servers. Each server sends a message only to its direct children to forward the management task (red arrows in Figure 3). For management tasks such as system monitoring, data from endpoints can be aggregated upward from the endpoints to the root. In this case, each server sends a message only to its parent (blue arrows in Figure 3). Thus, the root node does not handle excessive messages and hence hardly becomes a performance bottleneck.

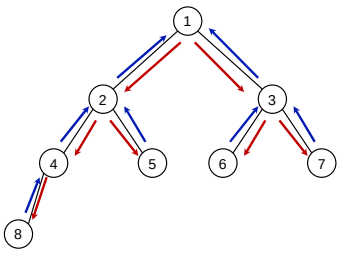

Figure 3: Management Server Tree for Scalability

2.3 Management Reliability Ring

Management servers constitute another logical network to provide failure resistance for reliable management. This network naturally forms a ring structure, named *Management Reliability Ring (MRR)*, through a primary-secondary relationship. Like the MST, the MRR is implicitly constructed by tree index number in the MSL. Figure 4 presents an example with k-level backup. A solid-lined node denotes a primary server and a dashed-lined node denotes a backup server. One primary server and *k* backup servers run on the same machine where those backup servers are for different primary servers running on different machines.

For failure detection and state synchronization, the primary server periodically sends a *keep-alive* message and *management information* to its first backup (secondary) server in another machine. The management information includes all information of all endpoints managed by each primary management server. For efficient communication, only the changes (delta) are sent piggybacking on the keep-alive message. Likewise, each backup server synchronizes with its next backup server. The arrow lines between servers in Figure 4 indicate this relayed communication process. For example, the primary server *1* synchronizes with the backup server *1'* (solid line), which then forwards the message to *1''* and then *1'''* (dashed lines) in a 3-level backup for very high reliability.

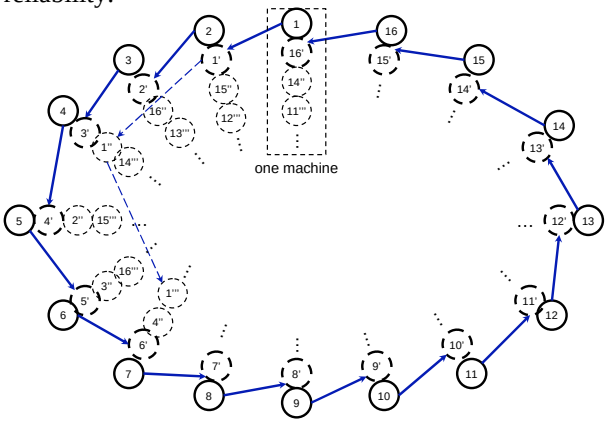

Figure 4: Management Reliability Ring for Failure Resistance

Note that we have a special placement algorithm for the backup servers. In Figure 4, the first backup server *1'* is placed on the next machine where primary server *2* is located (or *machine 2*). However, the second and third backup servers *1''* and *1'''* are not located on machine *3* and *4*. Instead of using the straight-forward

round-robin placement scheme, we use the following scatter placement scheme:

$$m(b_x(i)) = (x+(1+2+\ldots+i) + (n-1)) \% n + 1$$

, where $m(b_x(i))$ is the machine where the *i*-th backup server for primary server *x* is to be placed and *n* is the total number of machines. We assume one machine hosts only one primary server.

Our scatter backup placement scheme yields better load balancing than straight-forward round-robin scheme, where $m(b_x(i))=(x+i+(n-1))\%n+1$. This can be demonstrated by comparing the two schemes as to how many machines should fail to make all the k backup servers located on the same machine active. Note that the *i-th* backup server becomes active and takes over the management role only when its primary and all the prior *(i-1)* backup servers fail. With the round-robin scheme, all the *k* backup servers collocated on one machine become active when only *k* machines fail since the machines needed to fail to make the *k-th* backup server active cover all the machines needed to fail to run all the other *(k-1)* backup servers. In contrast, our scatter placement scheme requires at least *2k-1* machines to fail since we need *k* machines to run the *k-th* backup server and each of the other *k-1* backup servers has its previous backup server in a different machine. Therefore, the scatter placement scheme has the much lower probability of having many backup servers active at the same time on a single machine reducing load imbalance in case of serious failures.

There is a trade-off between providing high reliability with many levels of backup servers and efficient use of management resources. Since our scheme is general for any arbitrary level for backup servers, this is left to users as a configurable parameter. In our prototype, we choose a single backup server and call it a secondary server. It is reported that average availability of a single server can be achieved up to three nines (99.9%) [10]. With one secondary server, the management server availability can be calculated as $1 - (1 - 0.999)^2 = 99.9999\%$, six nines, which seems more than sufficient for most cases.[1]

In the presence of failure of the machine where the primary server is running, the secondary server becomes active to handle management tasks. For the process of this take-over, there are three states for a management server: PRIMARY, SECONDARY, and SECONDARYWAIT. A primary server starts in PRIMARY state while its secondary server starts in SECONDARYWAIT state. When the primary server fails, the secondary server switches over its role by changing its state from SECONDARYWAIT to SECONDARY. When the primary server recovers, the secondary server propagates updated management data to the primary server and returns to the SECONDARYWAIT state. An alternative is to put the recovered primary server in wait state and keep the secondary active. However, our policy enforces the secondary server to release its active role to the primary server since it is not desirable to have two servers (one primary server and one secondary server) active at the same time on the same machine while the machine where the primary server recovered just sits idling.

[1] This observation considers only hardware failure. If software failure and maintenance time is added, the effective availability can be much lower.

2.4 Management Server List

Aforementioned management networks, Management Server Tree and Management Reliability Ring, are constructed implicitly based on a globally shared data structure, called *Management Server List (MSL)*. MSL contains the basic information of all the servers in the management network, such as the ID and IP address of the currently active servers and the alternative servers. During normal operations, an active server is in PRIMARY state, and an alternative server is in SECONDARYWAIT state. When an active server in PRIMARY state fails, the active server and the alternative server are switched in MSL, and the active server turns into SECONDARY state, indicating the secondary server has taken over the management role. It also contains a tree index number which is used to construct the MST. An example of MSL is shown in Table 1. Whenever there is a change in the server network, the root server in the tree structure modifies the MSL and propagates the updates down the tree to the other management servers. Thanks to this globally shared management server index, users can manage any endpoints by connecting to any server in the network.

Tree Index Number	Active Server ID	Active Server IP	Alternative Server ID	Alternative Server IP
1	A	1.1.1.1	A’	2.2.2.2
2	B	2.2.2.1	B’	3.3.3.2
3	C’	4.4.4.2	C	3.3.3.1
…	…	…	…	…

Table 1: Management Server List

3. Mechanism

In this section, we first introduce and describe in detail a suite of new server operations that can expand or contract the management capacity in a scale-out

fashion. Then, we describe a set of simple and representative management tasks and how they are accomplished in a distributed manner in Blue Eyes.

3.1 Management Server Operations

We define a set of server operations to maintain the distributed server network. In Blue Eyes, a system administrator can add a new management server to an existing server network or merge two existing independent server networks into one. With these add and merge operations, the management server network can be expanded to handle more managed endpoints. Likewise, retiring management servers also can be deleted from a network to scale down the management system. On the other hand, to achieve high availability of system management, a fail-over operation is provided. Unlike add and delete operations, this operation is triggered implicitly. A secondary server automatically notifies the root server of the primary server's failure, so that the root server modifies MSL reflecting the change of the network where the secondary server takes over the failed primary server. A recovery of the primary server is also automatically reported to the root server and processed in a similar way. The root server balances the number of endpoints over the management servers by relocating some endpoints when there is a change in the management server network.

The *balance* operation, which is executed at the end of other operations that change the MSL, works as follows: the root server of MST reads the number of endpoints in each server and assesses the balance of management workload. If the imbalance exceeds a threshold, the root server sends the relocation request to other management servers to shed the load. The relocation request includes the number of endpoints to be relocated and a destination server. The servers which receive the relocation request pick the requested number of endpoints and move them to the specified destination server.

The *add* and *merge* operations share the same mechanism. The only difference is that *add* is used to add a single empty management server while *merge* is to merge two different server networks combining endpoints of each network into one network. In other words, *add* is equivalent to *merge* for joining an empty single server to an existing network. For a *merge* operation, a management console first sends the *merge* request of two management networks, one as a base and the other as a joining network. The request is forwarded to the root server of the target network, which modifies the MSL following the request and spreads it down to each server constructing a new expanded tree having the merged server network. The MSL of the joining network is added to the end of the MSL of the base network. The last server, the one with the largest tree index number in the MSL, of each network sends management information for their new secondary servers. Finally, the root server triggers the balance operation by sending relocation requests to each server. Figure 5 shows an example of merging the network Y into the network R. X and Z send their management information to new secondary servers.

The *delete* operation operates as follows: when a console issues a deletion of a server, the request is forwarded to the root server and the MSL is modified accordingly. Then, to adjust the MRR, the predecessor of the deleted server in the ring sends management information to its new secondary server which is to be located on the successor of the deleted server in the ring. Finally, the root server triggers the balance operation. An example of the delete operation is illustrated in Figure 6. The server W is deleted, and the server R sends the management information to its new secondary server.

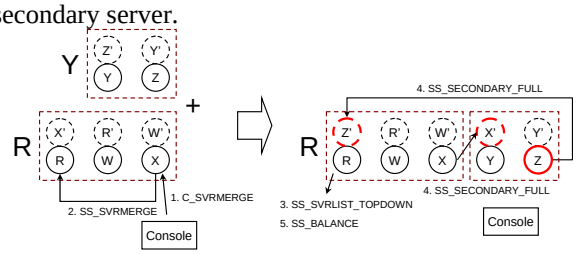

Figure 5: *add* and *merge*

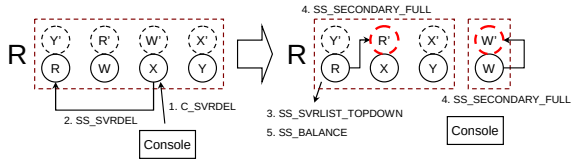

Figure 6: *delete*

To achieve system reliability, a server network must be self-organized when a server fails or recovers from failures. When a primary server fails, its secondary server detects the failure after a timeout of keep-alive messages from the failed server and notifies the root server. Then, the secondary server takes over the job and turns into SECONDARY state from the initial SECONDARYWAIT state. The root server modifies the MSL by switching the active server and the alternative server for the entry, and propagates the updates down to each server through the new tree topology. Figure 7 shows an example of the *fail-over* operation. The primary server W fails, and its secondary server W' takes over W. For the *recovery* operation, the root server is contacted in the same way to restructure the network. In Figure 7, when the failed W comes back alive, it takes over W', and W' goes

back to the SECONDARYWAIT state. It is possible not to switch back when the original primary server recovers. There is a trade-off between switch-over overhead and load imbalance. If the recovered primary server does not take back its management role, the machine hosting that server will not actively participate in management whereas the machine hosting the currently active secondary server can suffer by taking two roles. We chose to make the recovered primary server take over for management load balancing as this fail-over and recovery does not frequently occur and hence the switch-over overhead can be amortized.

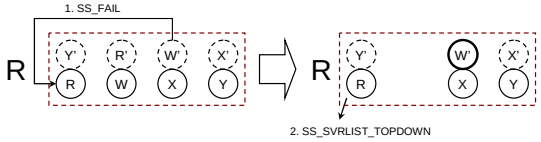

Figure 7: *fail-over*

3.2 Management Tasks

There are five basic types of management tasks that typical system management solutions support: *discovery*, *collect_inventory*, *view_inventory*, *software_distribution*, and *event_action*.

With these operations, an administrator can discover endpoint systems to manage given an IP address range, collect inventory information of discovered managed endpoints, or view collected inventory information, which includes hardware, operating system, network connectivity, and installed software information. Software or its updates can be installed on managed endpoints in a distributed and parallel manner through management servers. System administrators also can specify events to monitor and corresponding actions to take. For example, for the event "CPU usage of endpoint X becomes greater than 70%", an action can be set to "report the event to an administrator by email."

For the discovery task in Blue Eyes, a console sends the discovery task with a specific IP range and, optionally, a specific sub-tree, to the connected management server (say X). By allowing a specified sub-tree in the MST to run the discovery task, proximity of endpoints to nearby management servers in the network can be exploited. In this case, the discovered endpoints are marked not to be relocated to the servers outside the sub-tree during the balance operation. X first divides the IP range evenly into sub-ranges and assigns each range to each server in the specified sub-tree. Then, X forwards the discovery task with the divided IP ranges to the root server R of the sub-tree. R forwards the discovery task to all the other servers in the sub-tree down through the tree topology in the MST. Finally, each server independently discovers endpoints in the assigned IP range in parallel. Note that this local discovery procedure leverages the old traditional single server discovery implementation without changes. Figure 8 shows an example of the discovery task. The discovery task is executed under all the five servers through the root server, so that the IP range is divided into five sub-ranges.

The collect_inventory task is used when an administrator wants to collect more detailed information, such as resource usage and installed software, for the discovered endpoints. The algorithm is very similar to the discovery task algorithm except that the task carries endpoint IDs rather than IP ranges. The root server of the specified sub-tree in the MST forwards the collect-inventory task to relevant management servers. Then, each server collects inventory information of endpoints given the endpoint ID list and stores them locally.

The view-inventory task is used to aggregate and view inventory information of distributed endpoints. When a console sends the view-inventory task with an endpoint ID list and a specific sub-tree to the connected server X, X forwards the task to the root server R of the sub-tree. Then, R broadcasts the task to all the other servers in the sub-tree down the tree topology. Unlike collect_inventory, the view_inventory task performs extra work to aggregate inventory information using the tree topology. Each server aggregates collected inventory information from the management servers in a bottom-up manner. Finally, R sends the full inventory information to X, which then forwards it to the console. Figure 9 shows an example of the view-inventory task which lists the inventory of all endpoints in the entire system.

The software_distribution task is used when a software application or its updates need to be installed on specific endpoints. The first phase of the mechanism is similar to that of collect_inventory. The software_distribution task with an endpoint ID list, a specific sub-tree, and the software installer file URL is forwarded to servers through the tree topology. Then, each server sends the software installer URL to the specified endpoints in the task, and the endpoints download and install the software.

The last basic operation is *event_action*. A user can specify an event to monitor and set a corresponding action for endpoints. This task operates as follows. In Figure 8, a console sends the event_action task to the connected server Z. Then, Z specifies the event-handling server X, a root of the smallest sub-tree covering servers related to the event. For example, suppose the event is "CPU usage of endpoint EZ1 is higher than 80% and Disk usage of endpoint EW1 is higher than 80%." X distributes sub-events to the

relevant servers that cover the endpoints specified by the sub-events. Then, Z, the server managing EZ1, starts to monitor its CPU usage, and W, the server managing EW1, starts to monitor its disk usage. When Z or W detects the sub-events, it sends the information to the event-handling server X. X evaluates the event condition with the sub-event information. If the condition is evaluated true, X triggers the action specified by users, e.g. emailing an administrator about the event occurrence.

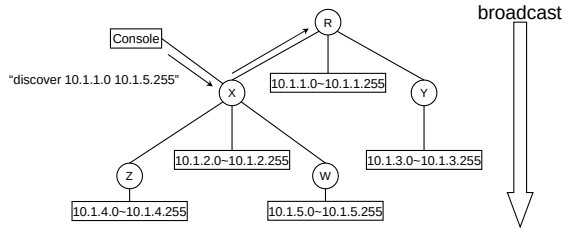

Figure 8: *discovery*

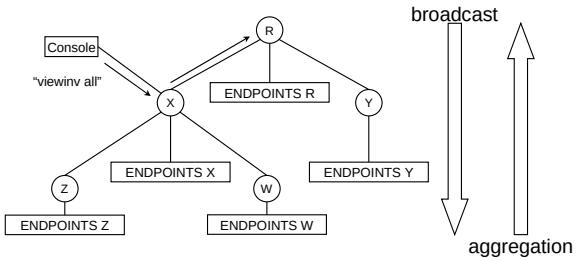

Figure 9: *view_inventory*

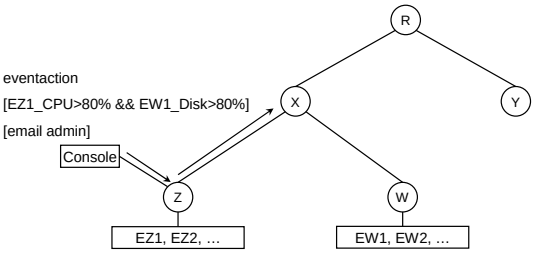

Figure 10: *event_action*

In a tree structure, the root can typically suffer a workload and communication bottleneck. As Blue Eyes exploits the tree structure for load distribution with an MST, we examine this potential problem. Regarding the network traffic between servers and endpoints, all the servers have almost equal workload because the number of endpoints is balanced over the servers through the balance operation overall or within a sub-tree if specified by users. For communication between servers, the root server handles more when it aggregates information from other servers if the task requires all the management servers to work. However, the tree structure is intended for the locality in group tasks where only a subset of endpoints are targeted and they are managed by a subset of management servers within a sub-tree. We believe this locality of task targets will prevent the root of an MST from being overloaded. In addition, the server-to-server traffic and task processing is far smaller than server-to-endpoint traffic and task processing, which will be shown in the evaluation section.

4. Implementation

We implemented a prototype of Blue Eyes with IBM Java 1.6.0 SDK. It is composed of three components: a *management server*, a *management console*, and a *management agent*. A management agent is a program running on each managed endpoint that communicates with a management server. Figure 11 illustrates the overall architecture of the Blue Eyes prototype. For simplicity, our prototype currently does not include a database component. Instead, all the information is handled and locally stored by management server. Each management server keeps the management information on the *management server list storage* and the *agent information storage*. A management server has a *management engine* that consists of a *console manager*, an *agent manager*, and a *server manager*. A console manager receives requests from a management console, processes them, and returns the results back to the console. An agent manager sends a management task to an agent, receives a response, and stores the result in the agent information storage. A server manager is the central part of our implementation of Blue Eyes. It maintains the MSL and handles all the server-to-server operations described in the last section such as *balance*, *add*, *merge*, *delete*, *fail-over*, and *recovery*. The server manager also sends the management information to a secondary server in order to provide reliability. Our prototype uses the lightweight UDP protocol between a management server and a management agent and the reliable TCP protocol for all other communications.

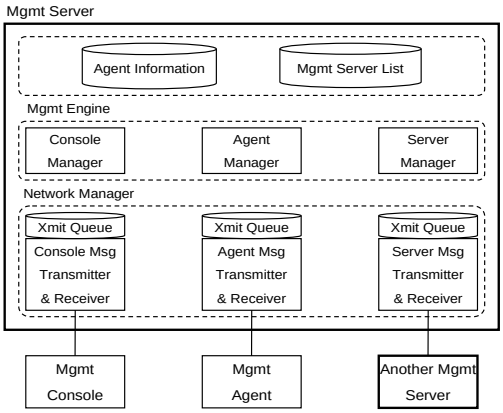

Figure 11: Implementation Architecture

5. Evaluation

In this section we evaluate Blue Eyes in terms of load balancing, performance, network traffic overhead, and reliability.

5.1 Experimental Setup

We used a Windows machine, the most popular platform on which to run a system management solution today, to host eight management server processes and one management console process. Each server process in the experiment setup represents a server machine in a real environment. Two Linux machines, each of which hosting 1024 agents, are used to represent 2048 managed endpoints totally.

Figure 12 shows our experimental environment. We assigned different port numbers to different management servers. For example, the first management server SVR1 uses TCP port 7001 to communicate with other management servers, UDP port 8001 to communicate with agents, and TCP port 9001 to communicate with a console. Even though secondary servers are not shown in Figure 12, each server has a corresponding secondary server in another process. Each agent on the same agent experiment machine uses its own IP address. All the agents use the same well-known port for discovery. All our experiments were scripted with Python to automate and repeat starting and finishing server processes and running server operations and management tasks

.

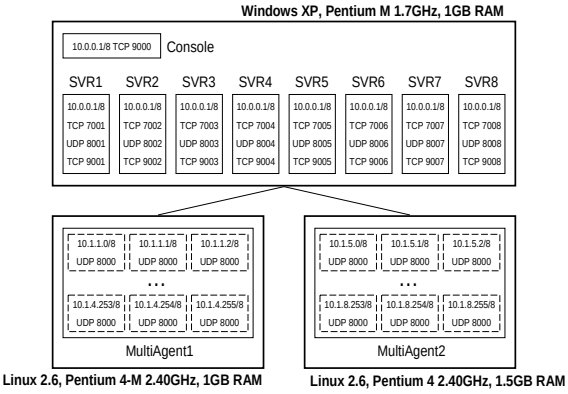

Figure 12: Experimental Environment

5.2 Load Balance

The first experiment demonstrates that the workload is distributed across management servers by balancing the number of endpoints. Figure 13 shows how the number of endpoints in each management server is balanced over time. We started with one server and discovered 256 endpoints at time 0. Then, we added an empty server and discovered another 256 endpoints every 60 seconds until reaching 240 seconds. At time 290 seconds, we merged a server network which contained three servers and 768 endpoints. The upper curve in the graph represents the total number of endpoints in the system; the number increases as new endpoints are discovered. The other curves on the bottom represent the number of endpoints managed by each server. This number is adjusted whenever a new server added, so that every server covers the same number of endpoints. For example, each of the three management servers, SVR1, SVR2, and SVR3 managed 256 endpoints before 170 seconds. Just after adding the fourth server SVR4, the balancing mechanism was executed by the root server. Each of the three old servers SVR1, SVR2, and SVR3 moved 64 endpoints to the new server SVR4, so that each server maintains the same number of endpoints, 192 endpoints. This experiment demonstrates that by balancing the number of endpoints to be covered, Blue Eyes distributes the management load over management servers.

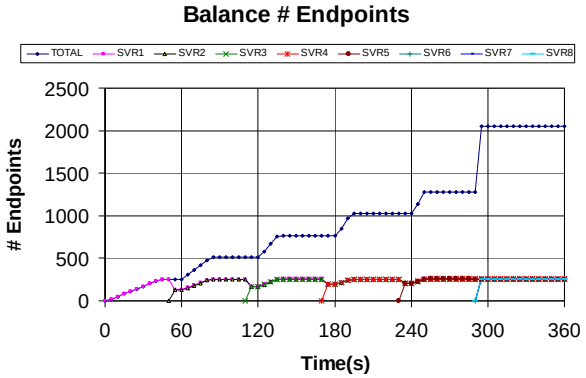

Figure 13: Load Balance

5.3 Performance

We conducted another experiment showing that Blue Eyes scales well in performance with addition of management servers. In this experiment, we varied the number of servers: one, two, four, or eight servers. For each setup, we started with all servers added and all 2,048 endpoints discovered, and then measured the progress of collect_inventory every five seconds. The eight servers took 50 seconds to collect inventory from 2048 endpoints, while a single server took 350 seconds. The speedup of the response time in this case is 7, which is quite close to the linear speedup. The difference is due to a communication delay over the tree structure for spreading the collect_inventory task

and aggregating the number of inventory-collected endpoints over servers. This shows the scalability of Blue Eyes is fairly good with near-linear speedup and the server-to-server interaction overhead is acceptable.

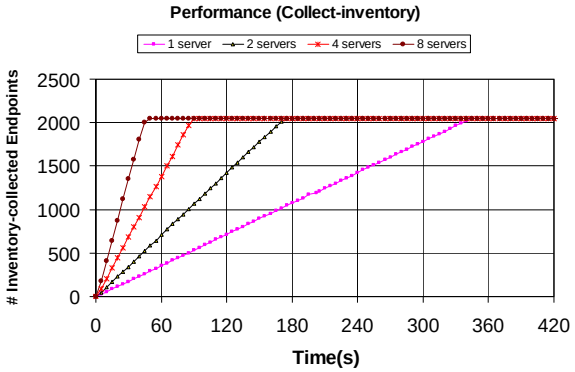

Figure 14: Performance

5.4 Network Traffic Overhead

We ran another experiment to analyze server-to-server interaction overhead in network traffic. In this experimental scenario, a user started with eight management servers, discovered all 2,048 endpoints at 20 seconds, collected inventory of all endpoints at 400 seconds, and viewed the inventory list of 512 endpoints in a sub-tree at 950 seconds. The experiment is finished at 1000 seconds. We categorized network traffic into three types of messages: server-to-endpoint, server-to-server for management tasks and server operations, and server-to-server for management information replication and keep-alive checking. Table 2 shows the total number of messages and the total amount of network traffic in the system for each category.

	total	**Server-to-Endpoint**	**Server – to-Server (Management)**	**Server –to-Server (Reliability)**
# msg (msgs)	69818	68110 97.5%	135 0.2%	1573 2.3%
Size (KBytes)	5791	4236 73.1%	171 3.0%	1384 23.9%

Table 2: Network Traffic (8 servers, 1000 seconds)

Note that server-to-endpoint traffic is independent of the number of management servers since each endpoint will receive the same number of requests and send out the same number of results. The results in Table 2 show that the messages between servers and endpoints dominate the management network traffic with more than 97% in the number of messages and 73% in total traffic amount. The server-to-server network overhead for management is sufficiently low in both of these metrics.

The extra network traffic overhead for reliability (24%) is not negligible. However, this is the trade-off between high reliability and resource efficiency. In this experimental setup, the keep-alive message period was set to 10 seconds, for very fast fail-over and recovery. This can be relaxed in practice, depending on the management availability requirement, which will further reduce the reliability overhead traffic. In addition, when more management servers are inserted with the addition of more systems to managed, extra network cables and subnets are typically added. This proportional expansion of network facility will lead to an acceptable constant bandwidth consumption among management servers when management network grows.

5.5 Fail-over and Recovery

The last experiment examines how the fail-over and recovery mechanisms operate in Blue Eyes. Figure 15 demonstrates that our system reliably manages endpoints in the existence of a server failure. Our failure scenario started with eight servers (SVR1~8) with 2,048 endpoints discovered and their inventory collected. We terminated the process running the primary server of the server SVR6 at 180 seconds. The secondary server of SVR6 detected the failure of the primary server at 210 seconds and took it over. We restarted the terminated process for SVR6 at 360 seconds to simulate the recovery. The curve on the top represents the number of endpoints available for management. The impact of the failure is localized and confined. For the 30-second between the failure and detection, only the local endpoints of SVR6 cannot be managed, but all the other endpoints covered by other servers are still manageable.

The curves on the bottom in Figure 15 show the CPU usage of the two processes[2], one (black solid curve) running the primary server of SVR6 and the secondary server of SVR5 and one (red dotted curve) running the secondary server of SVR6 and the primary server of SVR7. When the secondary server of SVR6 is active after fail-over between 210 seconds and 360 seconds, its CPU usage doubles whereas and the CPU usage of the process with the primary server of SVR6 stays at 0. This is due to increased management workload of that process as both primary SVR7 and secondary SVR6 become active. When the failed SVR6 primary recovers, it takes over SVR6's role

[2] This process represents a management server machine in the real use. In experiment, we use single machine and multiple processes, each of which represents a management server.

from the secondary and the CPU usages become balanced.

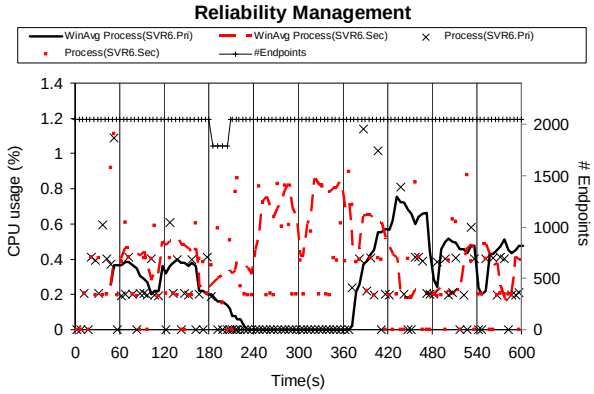

Figure 15: Reliable Management

6. Related Work

Many commercial system management solutions [1][2][3][4] have been developed to manage computing resources on different platforms. Those solutions support a set of common administration tasks such as discovery, inventory, and event notification. They also share a similar software architecture having only a single management server. This restricts the scalability. Their reliability model also relies on database recovery, which oftentimes extends the unavailable time.

There are several cluster monitoring systems that have a scalable architecture. Ganglia [5][6] is a scalable distributed monitoring system based on a hierarchical design for high performance computing systems such as clusters and Grids. Ganglia monitoring daemon (gmond) collects cluster information and provides it for the Ganglia Meta Daemon (gmetad) and a client. Ganglia is robust to failures of gmond but if the gmetad fails, the system becomes unavailable, resulting in a hole in reliability. Supermon [7] is another hierarchical cluster monitoring system that uses a statically configured hierarchy of point-to-point connections to gather and aggregate cluster data. A data concentrator aggregates information from node data servers running on cluster nodes. Cluster nodes are similar to managed endpoints in system management and data concentrators are similar to management servers. The data concentrators can build a hierarchical tree to support additional managed endpoints, but only the leaf data concentrators in the tree communicate with node data servers, whereas intermediate nodes in the Blue Eyes's MST covers its share of endpoints. Also, there is no fail-over mechanism for data concentrators, resulting in lack of reliability. PARMON [8] is a client/server cluster monitoring system that uses servers that export a fixed set of node information and clients that poll the servers and interpret the data. Servers in PARMON are similar to management agents in our system. Clients in PARMON correspond to management servers in our system, but they cannot be connected to one another in order to handle a large amount of managed endpoints, therefore PARMON does not offer a scalable solution.

Linux Virtual Server [11][12] provides a basic framework to build scalable and available network services. It has a three-tier architecture with a load balancer, a server pool, and back-end storage. The front-end load balancer simply forwards incoming packets to the server pool which then performs the work and manipulates data in the back-end storage. The state of concurrent connections is maintained by the load balancer and servers in the server pool are required to be stateless. This architecture with stateless servers is not suitable for existing system management solutions. A system management server frequently monitors the status of each endpoint and keeps them in memory. Requiring servers to keep all the frequent changes of endpoints in the shared back-end database is prohibitively expensive. The P2P research [13][14][15][16] has shown that a scalable and reliable system for storing and retrieving data can be built upon unreliable machines and networks. Nodes in a P2P network do not need a global map to create a highly scalable network. However, the complexity of managing a network in a P2P system is too high to be applied to a system management solution. In addition, machines used for management servers are usually more robust than the nodes joining and leaving in a P2P network since management servers are set up and maintained by enterprise administrators. We believe a hierarchical structure is sufficient to achieve low complexity and high scalability for a management server network.

Our previous research [9] describes a technique for testing and validating a commercial grade system management tool for thousands of managed endpoints. It uses "agent multiplication" to make one physical test machine appear as many managed endpoints to the management server, while maintaining all of the managed endpoints as distinct management agents. We leveraged this technology to evaluate Blue Eyes in a large scale environment with a small number of testing machines.

7. Conclusions and Future work

This paper presents and evaluates Blue Eyes, a scalable and reliable system management solution based on a multi-server architecture. We described how multiple management servers are organized to

effectively and reliably manage ever increasing endpoint systems requiring administration. Management workload is evenly distributed by balancing the number of endpoints covered by each management server. Management tasks are forwarded efficiently to endpoints through a tree structure (MST) of management servers. Secondary servers organized in a logical ring (MRR) help the system be reliable and highly available. We implemented a prototype of Blue Eyes with a core set of server-server operations and distributed mechanisms to support a basic management tasks. It is important to note that this implementation does not impose significant modification of management task code of the legacy management solutions. Our early experimental results, using the prototype, show that Blue Eyes can be expanded with balanced workload for endpoints, good performance characteristic for management tasks, and a small scale-out overhead. We also assessed how the system resource utilization changes over a fail-over process.

Currently, we are investigating adding automatic endpoint grouping based on proximity to management servers. By automatically grouping endpoints within a short distance and managing them under a certain management server of a sub-tree, IT administrators can manage endpoints even more efficiently exploiting locality. We are also expanding our experiments in two dimensions, 1) larger scale experiments and measurements with hundreds of thousands of endpoints, and 2) various types of management tasks that our experiment did not explore in this paper such as software_distribution and event_action.

References

[1] D. Watts, et al., Implementing IBM Director 5.20. SG24-6188, IBM Corp., April 2007, http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246188.html.

[2] Microsoft Corp., Microsoft System Management Server Planning Guide. 2008. http://www.microsoft.com/technet/prodtechnol/sms/sms2/proddocs/default.mspx?mfr=true.

[3] C. Cook, et. al., An Introduction to Tivoli Enterprise. SG24-5494, IBM Corp. http://www.redbooks.ibm.com/abstracts/sg245494.htm.

[4] HP, HP Systems Insight Manager Technical Reference Guide. November 2005. www.hp.com/wwsolutions/misc/hpsim-helpfiles/sys-book.pdf.

[5] M. Massie, et. al., The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing, Vol. 30, Issue 7, July 2004.*

[6] F. Sacerdoti, et. al., Wide Area Cluster Monitoring with Ganglia. *IEEE Cluster 2003 Conference, Hong Kong, December 2003.*

[7] M. Sottile, et. al., a high-speed cluster monitoring system. *Proceedings of Cluster 2002, September 2002.*

[8] R. Buyya, Parmon, a portable and scalable monitoring system for clusters. *Software—Practice and Experience 30 (7) (2000) 723–739.*

[9] K. Ryu, et. al., Agent Multiplication: An Economical Large-scale Testing Environment for System Management Solutions. *Workshop on System Management Techniques, Processes, and Services (SMTPS), April 2008.*

[10] W. H. Highleyman, Availabilty versus Performance. Sombers Associates, Inc., August 2007. http://www.availabilitydigest.com/public_articles/0208/availability_versus_performance.pdf.

[11] W. Zhang, et. al., Creating Linux Virtual Servers. LinuxExpo 1999.

[12] W. Zhang, Linux Virtual Server for Scalable Network Services. Ottawa Linux Symposium 2000.

[13] S. Ratnasamy, et. al., A Scalable Content Addressable Network. *Proceedings of the ACM SIGCOMM, Aug. 2001.*

[14] A. Rowstran, et. al., Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), November 2001.*

[15] I. Stoica, et. al., Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *Proceedings of the ACM SIGCOMM, August 2001.*

[16] B. Zhao, et. al., Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications, 22(1), January 2004.*