In: *Proceedings of the Sixth IEEE International Symposium on High Assurance Systems Engineering,* Boca Raton, Florida, October 2001, pp. 139-148.

The SASHA Architecture for Network-Clustered Web Servers*

Steve Goddard
Computer Science & Engineering
University of Nebraska—Lincoln
Lincoln, NE 68588-0115
goddard@cse.unl.edu

Trevor Schroeder
Media Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139-4307
tschroed@media.mit.edu

Abstract

We present the Scalable, Application-Space, Highly-Available (SASHA) architecture for network-clustered web servers that demonstrates high performance and fault tolerance using application-space software and Commercial-Off-The-Shelf (COTS) hardware and operating systems. Our SASHA architecture consists of an application-space dispatcher, which performs OSI layer 4 switching using layer 2 or layer 3 address translation; application-space agents that execute on server nodes to provide the capability for any server node to operate as the dispatcher; a distributed state-reconstruction algorithm; and a token-based communications protocol that supports self-configuring, detecting and adapting to the addition or removal of servers. The SASHA architecture of clustering offers a flexible and cost-effective alternative to kernel-space or hardware-based network-clustered servers with performance comparable to kernel-space implementations.

1. Introduction

The exponential growth of the World Wide Web, coupled with increasing reliance on dynamically generated pages, has left a large gap between the needs of high volume sites and the ability of web servers to satisfy that need. Into this gap have stepped a number of multi-computer solutions which attempt to solve the problem while utilizing, as much as possible, commodity systems. These multi-computer solutions tie a pool of servers together to create a *server cluster* with a central coordinator, which we call the *dispatcher*. The dispatcher provides a central point of contact for a cluster of web servers as shown in Figure 1.

Clustering has traditionally been implemented with specialized operating systems or with special-purpose hardware. However, in this paper, we present a Scalable, Application-Space, Highly-Available (SASHA) network-clustered web server architecture that demonstrates high performance and fault tolerance using application-space software and Commercial-Off-The-Shelf (COTS) hardware and operating systems. The use of COTS systems throughout the cluster allows us to take advantage of the price/performance ratio offered by COTS systems while still providing excellent performance and high availability. We combine our dispatcher with agents that execute on the server nodes to provide the capability for any server node to operate as a dispatcher node. This, combined with a distributed *state-reconstruction* algorithm, instead of the more typical *primary-backup* [2] or *active replication* [29] approaches for fault recovery, provides us with the ability to operate without a designated standby unit for the dispatcher. In addition to tolerating the loss of the dispatcher, the SASHA architecture is able to detect and dynamically adapt to the addition or removal of servers.

The rest of the paper is organized as follows. Section 2 discusses background and related work. Section 3 presents the SASHA architecture: an application-space dispatcher program, the TokenBeat protocol, application-space agents on the server nodes, and distributed state reconstruction. Section 4 examines the performance of a SASHA prototype under non-faulty operation, single-fault operation, and high-fault operation. Finally, Section 5 summarizes the SASHA architecture and our contributions.

2. Background and Related Work

All web server clustering technologies are transparent to client browsers (i.e., the client browsers are unaware of the existence of the server cluster). However, not all clustering technologies are transparent to the web server software. Early commercial cluster-based web servers, such as Zeus [36] and HotBot (based on an architecture proposed by Fox *et al.* [12, 13]) are, in many respects, continuations of the traditional approach to cluster-based computing: treat the cluster as an indissoluble whole rather than the lay-

*This work supported, in part, by grants from the National Science Foundation (EIA-0091530) and the University of Nebraska-Lincoln Center for Communication and Information Science and a contract with Flextel S.p.A.

In: *Proceedings of the Sixth IEEE International Symposium on High Assurance Systems Engineering,* Boca Raton, Florida, October 2001, pp. 139-148.

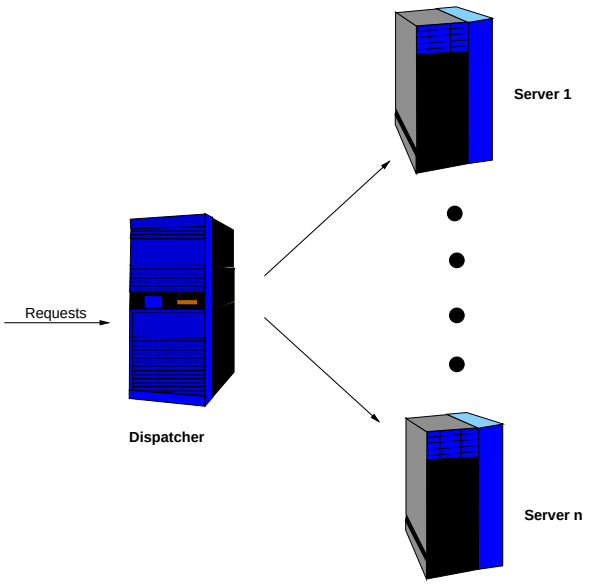

Figure 1. Conceptual View of a Typical Network-Clustered Web Server.

ered architecture assumed by (fully) transparent clustering. Thus, while transparent to the clients, these systems are not transparent to the server nodes, and require specialized software throughout the system.

For example, the architecture proposed by Fox *et al.* [12, 13] has a central point of entry and exit for requests, but nodes in the cluster are specialized to perform certain operations such as image manipulation, document caching, etc. There is a coordinator that organizes and controls the nodes in the servicing of client requests, which requires custom web server software. In a similar vein, the Zeus web server provides server clustering for scalability and availability, but each server node in the cluster must be running the Zeus web server, a specialized server software developed for this environment.

Network-clustering technologies are transparent to the server software. The various approaches to network-clustering of servers are broadly classified as: *OSI layer four switching with layer two packet forwarding* (L4/2); *OSI layer four switching with layer three packet forwarding* (L4/3); and *OSI layer seven* (L7) *switching with either layer two packet forwarding* (L7/2) *or layer three packet forwarding* (L7/3) clustering. These terms refer to techniques by which the servers in the cluster are tied together. In addition to these network layer classifications, implementations may be application-space, kernel-space, or based on special-purpose hardware. This rest of this section briefly describes these classifications and implementation options. See [31] for more detailed information on these classifications.

2.1. Layer 4 Switching With Layer 2 Address Translation

The majority of the server network-clustering devices dispatch messages to server nodes in the cluster using OSI layer 4 switching with layer 2 address translation. Commercial products in this category include IBM's eNetwork Dispatcher [18] and Nortel Networks' Alteon ACEdirector [24]. Research prototypes in this category include ONE-IP developed at Bell Labs [11] and LSMAC from the University of Nebraska-Lincoln (UNL) [16].

In L4/2 clustering, the dispatcher and a set of servers are all assigned a shared cluster address. Incoming traffic for the cluster address is routed to the dispatcher (this may be done via static ARP entries, routing rules, or some other mechanism).

Upon receiving the packet, the dispatcher examines it and determines whether it belongs to a currently established connection or is a new connection. If it is a new connection, the dispatcher utilizes its load-sharing policy to choose a server to service the request and records the connection in a map maintained in the dispatcher's memory. The MAC address of the packet is then rewritten to be that of the chosen server and sent to that server.

The server receives the packet and since it has an interface configured with that IP address, processes it as a packet destined for itself. Reply packets are sent out via the default gateway. Upon termination of a TCP session, the dispatcher deletes the connection.

This technique is extremely simple and provides high throughput as the two halves of the TCP stream are decoupled: the dispatcher processes only a small amount of incoming data while the large volume of return data is sent straight from the server to the client. Additionally, no TCP checksum recomputations are required, only frame checksums, which are done by the network interface hardware.

2.2. Layer 4 Switching With Layer 3 Address Translation

Server clustering based on Layer 4 switching with Layer 3 address translation is also known as "Load Sharing Using Network Address Translation (LSNAT)", and is detailed in RFC 2391 [33]. Commercial products in this category include Cisco's LocalDirector [7] and research prototypes include Magicrouter from Berkeley [3] and LSNAT from UNL [16]. Magicrouter was an early implementation of this concept based on kernel modifications) [3] and LSNAT from UNL is an example of a non-kernel space implementation [16]. L4/3 clustering shares the basic layer 4 clustering concept with L4/2 clusters, but differs from the L4/2 approach in many significant ways.

In an L4/3 system, each server in the server pool has a unique IP address. The dispatcher is usually the sole machine assigned the cluster address. Incoming traffic is, as

before, compared with a map of existing connections. If the traffic belongs to one of these, the destination IP address is rewritten to the address of the chosen server and IP and TCP checksums are recomputed in software. This packet is then sent to the particular server servicing the request. In the event that it does not belong to an existing connection, a server is chosen using the load-sharing policy and the packet is processed as just described.

The server chosen to service the packet then receives the packet, processes it and sends a response *back to the dispatcher*. Without changes in the network protocol, the server operating system, or device drivers, the packet must be sent back to the dispatcher since the reply is sent with a source address different from the address the client originally sent its request to. The dispatcher changes the source address from that of the responding server to the cluster address, recomputes checksums, and sends the packet to the client.

2.3. Layer 7 Switching

The final dispatching method, also known as content-based routing, operates at Layer 7 of the OSI protocol stack. Commercial products in this category include Cisco's CSS 11000 switch [8] and Nortel Networks' Alteon Personal Director (PCD) [25]. There are many research prototypes in this category, with most of them focusing on providing some form of Quality of Service (QoS) based on the content of the request (e.g., [1, 5, 6, 9, 21, 26, 27])

In an L7 cluster, as in L4 clustering, a dispatcher acts as the single point of contact for the cluster. Unlike L4 clustering, however, the dispatcher does not merely pass independent packets on to the servers servicing them. Rather, it accepts the connection, receives the client's L7 request, and chooses an appropriate server based on that information.

After choosing a server, either layer 2 or layer 3 packet forwarding is used. In the event that layer 2 packet forwarding is used, the dispatcher must have a means to inform the target web server of the connection already established. LARD from Rice University [26] does this using a modified kernel that supports a connection hand-off protocol on all of the server nodes. If layer 3 switching is chosen, the dispatcher essentially connects to the back-end server, makes the request, and relays the data to the client.

L7 clustering has the benefit that server nodes may be chosen on the basis of message content in the application layer protocol. For example, it may be advantageous to choose one or two high-performance servers to service CGI requests while leaving the lower-performance systems to serve static HTML content. The web document tree may also be split into disjoint subtrees which are then assigned to the individual servers. In this way, we can increase the locality of the data that each server serves and thus improve the performance of the system as a whole. Layer 7 switching is often combined with caching on the dispatcher to decrease the load on the server nodes.

2.4. Implementation Choices

In addition to the choice of one of three major clustering approaches, implementors are faced with the choice of where their dispatcher should be implemented: in application space, in kernel space, or in specialized hardware. Most implementors have chosen either kernel-space or hardware-based solutions for performance reasons.

With a kernel-space implementation, such as eNetwork Dispatcher or LocalDirector, the incoming traffic does not need to be copied in and out of application space. Additionally, sending and receiving packets do not cause expensive mode switches from user-mode to kernel-mode. The hardware used, however, is more-or-less commodity hardware: an RS/6000 in the case of eNetwork Dispatcher [18] and a custom Pentium II PC in the case of LocalDirector [7].

Hardware-based implementations, such as the CSS1100 line of switches from Cisco, often improve performance by an order of magnitude. This is because many of the repetitive tasks, such as checksum recalculation and address translation, can be handed off to specialized hardware.

To date, however, there seems to be little interest in purely application-space solutions. We believe that this ignores some of the distinct advantages of application-space solutions: flexibility, portability, and extensibility. Thus, work in the UNL Advanced Networking and Distributed Experimental Systems (ANDES) laboratory has focused on application-space solutions [15, 16, 28, 31, 35].

Recently, we have begun to address the issue of fault-tolerance in network-clustered servers. Almost all of the network-clustering dispatchers referenced here support the loss of server nodes in the cluster. They simply quit sending new requests to faulty server nodes. All connections that had been active on the server node at the time of the fault are lost, but the cluster itself remains operational. It is usually assumed that the client will simply send the request again and the dispatcher will assign the request to a healthy server.

In this work, we address the more difficult problem of tolerating benign faults in the dispatcher by developing the SASHA architecture for network-clustered web servers based on Layer 4 switching. It provides the unique capability of operating without dedicated standby units while still providing high availability and high performance.

3. The Architecture

The Scalable, Application-Space, Highly-Available (SASHA) architecture for network-clustered web servers consists of the following components: an application-space dispatcher program, the TokenBeat protocol, application-space agents on the server nodes, and distributed state reconstruction. Our use of COTS systems throughout

the cluster allows us to take advantage of the excellent price/performance ratio offered by COTS systems while still providing excellent performance and high availability. We combine our dispatcher with agents that execute on the server nodes to provide the capability for any server node to operate as a dispatcher node. All components of the SASHA architecture execute in application-space and are not tied to any particular hardware or software. At any given time, one computer operates as a dispatcher and the rest as server nodes. While it is possible that some nodes might be specialized (i.e., lacking the ability to operate as a dispatcher or lacking the ability to operate as a server), we assume any computer can be either a server node or the dispatcher for this presentation.

By choosing an application-space solution, we can take advantage of low-cost commodity hardware and software to build an n fault-tolerant system with enough performance to satisfy the demands of most commercial sites. The use of COTS systems also provides a degree of freedom and heterogeneity that non-commodity (software or hardware) cannot provide.

The SASHA architecture is capable of providing comparable performance to in-kernel software solutions while simultaneously allowing for easy and inexpensive scaling of both performance and fault tolerance. Moreover, unlike commercial network-clustering products, the SASHA architecture does not require a hot-standby node to tolerate the fault of the dispatcher. In the SASHA architecture, one of the server nodes takes over the role as dispatcher when the loss of the previous dispatcher is detected. Thus, the SASHA architecture provides graceful performance degradation in the loss of any node in the cluster, including the dispatcher.

The rest of this section describes the SASHA dispatcher; TokenBeat, a network protocol developed to provide group messaging capabilities along with basic fault detection; server or cluster fault detection and recovery using application-space agents; state reconstruction, an alternative to traditional active state replication or primary-backup approaches; and the flexibility that SASHA offers in high-fault scenarios.

3.1. The Dispatcher

The SASHA dispatcher is an application level program running on a commodity system. More specifically, it is a Layer 4 switch using layer 2 or layer 3 address translation. (We have also implemented a layer 7 application-space dispatcher that uses the TokenBeat protocol to detect and recover from faults. However, our distributed state reconstruction algorithm will not work with layer 7 dispatchers since the server nodes do not know the identity of the clients.) In this work, we present and evaluate an L4/2 instance of the SASHA dispatcher.

In developing the SASHA architecture, one of our chief goals was portability. This allows the end-user maximum flexibility in designing their system. Anything from a low-end PC to the fastest SPARC or Alpha systems may be used. Our instance of the SASHA architecture is written using the packet capture library, `libpcap` [20], the packet authoring library, `Libnet` [10], and POSIX threads [19]. This provides us with maximum portability, at least among UNIX compatible systems. As an added benefit, the use of `libpcap` on any system which uses the Berkeley Packet Filter (BPF) [20], eliminates one of the chief drawbacks to an application-space solution. BPF only copies those frames which are of interest to the user-level application and ignores all others, reducing frame copying penalties and the number of times we must switch between user and kernel modes.

The L4/2 SASHA dispatcher prototype we developed operates largely as described in Section 2.1 and summarized in Figure 2. We create a *virtual IP* (VIP) address for the cluster that is shared by all nodes in the cluster. We also create a *virtual MAC* (VMAC) address for the cluster and configure the router to forward all cluster addressed packets to the subnet shared by the dispatcher and server nodes. When the SASHA dispatcher begins, it places the NIC in promiscuous mode and uses `libpcap` with a filter to retrieve all L4/2 messages destined for the VMAC address. Once received, the messages are processed and forwarded to a server node, as described in Section 2.1.

The use of a VMAC address simplifies recovery from a dispatcher fault. When one of the SASHA agents executing on the server nodes detects a crash of the dispatcher, it calls for a TokenBeat ring purge (described in Section 3.2), which automatically triggers a TokenBeat ring reconstruction around the faulty dispatcher. After a healthy node is elected (as described in Section 3.3) to be the new dispatcher, the SASHA agent reconstructs the cluster state using the algorithm described in Section 3.4. Next, the SASHA agent launches the dispatcher program, which places the NIC in promiscuous mode and listens for packets addressed to the VMAC (just as the original dispatcher did).

3.2. The TokenBeat Protocol

To provide fault-tolerant operation, we developed the TokenBeat protocol [30]: an extremely lightweight, non-reliable, token-passing, group messaging protocol. This is in contrast to protocols such as Totem [23] or Horus [34] which are designed to be general purpose, reliable, large-scale token-passing group messaging protocols. We wanted a protocol that requires very few network, processing, or memory resources. Moreover, the protocol needed to be easily and closely integrated into an application specific role (to remain simple and lightweight).

TokenBeat is not a general-purpose network protocol,

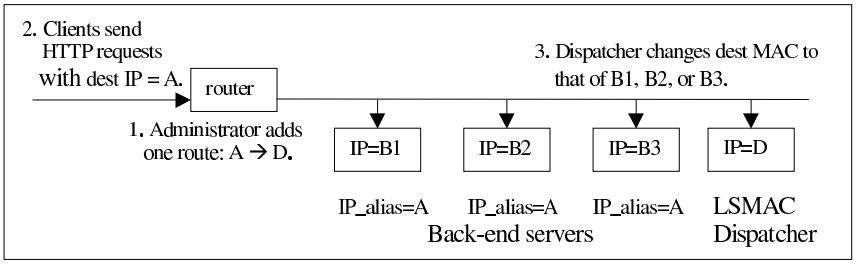

Figure 2. SASHA dispatcher implementation in a LAN environment.

such as IP, but rather designed to be modified and extended to support specific applications. Its emphasis is on simplicity and low bandwidth. The simple nature of the protocol minimizes the impact in terms of application complexity and computational expense. The low bandwidth requirement of TokenBeat supports deployment in bandwidth-constrained environments, such as embedded systems or—as in SASHA's case—a high utilization network. The remainder of this section provides a high-level overview of the TokenBeat protocol. See [30] for a detailed description of the protocol.

The SASHA dispatcher node and the server nodes compose a logical ring, which we refer to as the TokenBeat ring. The TokenBeat ring master, typically the dispatcher, circulates a self-identifying heartbeat message. As long as this message circulates, the TokenBeat ring is assumed to be whole and thus the system is assumed to be fault-free. As we will see in the next section, this greatly restricts the *types* of faults which we can tolerate. With a few exceptions, no TokenBeat messages are sent directly to the recipient. Rather, they are relayed through intermediate nodes. This is similar to most token-passing protocols, and is done to provide constant fault detection and quick recovery. Unlike most token-passing protocols, TokenBeat allows nodes to create new tokens (packets) and send them on with their own message payloads rather than waiting to receive the current token (packet). This allows for out-of-band messaging in critical situations such as node failure.

If a new server comes online, it broadcasts its intention to join the ring. It is then assigned an address and inserts itself into the ring. If a server crashes, the logical ring is broken. Messages do not propagate down stream from the crashed node. This break is detected by the lack of messages, as mentioned before, and a *ring purge* is forced, which causes all nodes to leave the ring and reenter just as they did upon starting up. The ring purge and reconstruction allows the ring to re-form without the faulty node. Figure 3 shows a logical representation of this. On the left, we see a four node ring operating normally. In the middle, node four has crashed, breaking the ring. Finally, node one declares a purge and the ring reforms without node four, as seen on the right.

The TokenBeat messages may be sent using the LAN carrying the client/server traffic being processed by the cluster. Alternatively, a separate LAN can be used just for fault detection and recovery. The former configuration allows for easy integration into existing systems while the later configuration provides for faster fault detection and reconfiguration. In this work, we evaluate the performance of a SASHA web server in which the TokenBeat messages must compete with client/server traffic on the same LAN—providing a worst case evaluation.

3.3. Fault Detection and Recovery using Application-Space Agents

In this work, we assume that all faults are *benign*. That is, we assume that all failures cause a node to stop responding and that this failure manifests itself to all other nodes on the network. This behavior is usually exhibited in the event of operating system crashes or hardware failures. Note that other fault modes could be tolerated with additional logic, such as acceptability checks and fault diagnoses. For example, all HTTP response codes other than the 200 family imply an error and the server could be taken out of the active pool until repairs are completed.

It is important to note that when we speak of fault-tolerance, we are speaking of the fault-tolerance of the *aggregate system*. When node failures occur, all requests in progress on the failed node are lost. No attempt is made to complete the in-progress requests using another node. For most HTTP traffic, this is too much overhead for the value returned.

In the event that a server (including the dispatcher) goes off-line, the TokenBeat ring is broken, heartbeat messages stop circulating, the break is detected by application-space agents on the server nodes, and a ring purge is forced. This detection is based on a configurable timeout interval. Without the ability to bound the time taken to process a message, this interval must be experimentally determined. Our experience shows that at extremely high loads, it may take an application-space agent more than a second to receive, process, and pass on TokenBeat packets. (This time can be reduced if we run the SASHA agents at a higher priority, but we wanted to evaluate the architecture with an unmodified

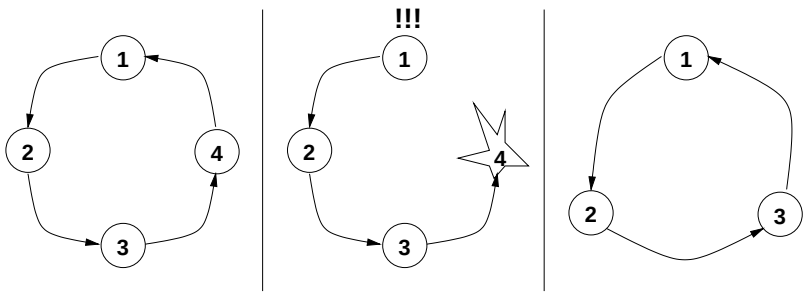

Figure 3. TokenBeat ring: normal operation, error detection, and the newly formed ring after ring purge (recovery).

system configuration running the application programs at default levels.) For our tests, the timeout threshold was set to 2,000 ms. Upon detecting the ring purge, the dispatcher marks all servers as 'dead'. As the servers reinsert themselves into the ring, their status is changed to 'alive' and they are once more available to service client requests. In this fashion, we automatically detect and mask server failures.

In the event that the dispatcher goes off-line, the TokenBeat ring is, as before, broken and a ring purge is forced. After the ring has been reconstructed (the ring is deemed reconstructed after a certain interval has expired, 2,500 ms, in this case), the agent on the server with the smallest TokenBeat address will notice the absence of the dispatcher's self-identification messages. It will then elect a new dispatcher from among eligible nodes. Any of the various election algorithms from the literature may be used for the election (e.g., [14, 17, 32]). However, in the SASHA architecture, we prefer to use a less dynamic algorithm. For example, in a homogeneous system, the machine with the lowest address, the ring master, is 'elected' to be the dispatcher. In a heterogeneous environment of nodes with different capabilities, the ring master might not have the capability to act as a dispatcher. In such a case, we choose the machine with the lowest address *and* the capability to act as the dispatcher. If the old dispatcher rejoins the ring at a later time, the two dispatchers will detect each other and the one with the higher address will abdicate and become a server node. Of course this mechanism may be extended to support scenarios where more than two dispatchers have been elected, such as in the event of network partition and rejoining. We assume that in the case of network partitioning, only one of the partitions will receive messages from the router. Thus, the election of two dispatchers in a partitioned network will not result in packets being processed by two different servers.

3.4. State Reconstruction

To date, the most popular method to provide fault tolerant operation in a network-clustered server has been to use hot-standby units with either active replication [29] or the primary-backup [2] method of achieving state replication in the standby unit. In the active replication approach, the secondary unit is at all times, an exact replica of the primary unit. In the primary-backup approach, the primary sends periodic *state update* messages to the standby (backup) unit. The length of the periodic update interval determines the accuracy of the state in the standby unit. In both sate replication approaches, communication between the primary and standby units is typically achieved with a special out-of-band interconnect, such as LocalDirector's failover cable [7]. Under normal (i.e., non-faulty), operation, the secondary unit performs no useful function. Instead, it merely tracks the setup and teardown of (potentially) thousands of connections per second.

By contrast, SASHA utilizes a novel distributed state reconstruction algorithm based on two observations.

1. The state of web servers is relatively small but extremely dynamic. At any given time there are only a few thousand connections established to the back-end servers.

2. Each of the server nodes in an L4/2 or L4/3 network-clustered server know the identity of the client they are serving.

Under these conditions, it is only marginally slower to *reconstruct* the state *during failure recovery* than to use replicated state. Our state reconstruction approach is very different from both the traditional approaches of replicating state and the *soft state* reconstruction approach employed by Fox *et al.* [12, 13] where cached state information is periodically updated with state update messages. According to [12], "cached stale state carries the surviving components through the failure. After the component is restarted, it gradually rebuilds it soft sate . . . "

When a dispatcher comes online, it uses the messaging services provided by TokenBeat to query the SASHA agents executing on the server nodes for a list of active connections. These are then entered into the dispatcher's connec-

tion map to *reconstruct the state of the cluster*. The new dispatcher then continues operation as normal. Additionally, when a new server joins (or a previously dead server comes to life), it is queried for connection state information. In this fashion, we avoid the need for active state replication and dedicated standby units.

3.5. Flexibility In High Fault Scenarios

SASHA's architecture provides a very important advantage over traditional network-clustered servers: flexibility in high fault scenarios. While specialized (kernel or hardware) solutions may provide fault tolerance (usually one dispatcher fault and multiple server faults), it is at the expense of cost efficiency. The introduction of a standby dispatcher unit increases the cost of the cluster but does not improve the performance of the system.

The SASHA architecture is more efficient in that it provides the capability of adding a high degree of fault tolerance without requiring dedicated standby units. The potential for each server to act as a dispatcher means that the available level of fault tolerance can be equal to the number of server nodes in the system. Under normal operation, one node is the dispatcher and other nodes operate as servers to improve the aggregate performance of the system. In the event of a fault, even multiple faults, a server node may be elected to be the dispatcher, leaving one fewer server nodes. Thus, increasing numbers of faults gracefully degrades the performance of the system until all units have failed.

The fault tolerance and recovery model of the SASHA architecture is in marked contrast to the behavior of hot-standby-based models where the system maintains full performance until the primary and all standby dispatcher units fail (at which point the entire system fails, even though there may be server nodes still operating). Increasing the reliability of our system also increases the performance of the system. In the event that all nodes but one has failed, this node may detect it and rather than becoming the dispatcher, operate as a stand-alone web server.

4. Experimental Results

This section evaluates experimental results obtained from a prototype of the SASHA architecture based on an L4/2 dispatcher. We consider the experimental setup as well as the results of tests in various fault scenarios under various loads. The reader will note that the experiments were done on "relatively old computers." This was intentional. We have found that the performance of SASHA clusters is limited by 1) the dispatcher, 2) the number and capability of the servers, and 3) LAN bandwidth. We have shown that even with "old" computers, LAN bandwidth is the limiting factor in performance with some client access patterns [16]. We have compared performance experiments in which the dispatcher was the bottleneck and found that the cluster performance increased linearly with respect to the increased capability of the dispatcher. In the results presented here, the dispatcher was not the bottleneck; the servers were. We think this experiment best highlights both the strengths and the weaknesses of the SASHA architecture.

4.1. Experimental Setup

The experimental setup is as follows.

- Clients: Each client node was an Intel Pentium II 266 with 64 or 128 MB of RAM running version 2.2.10 of the Linux kernel. In all test cases, there were 5 client machines.
- Servers: Each of the five server nodes was an AMD K6-2 400 with 128 MB of RAM running version 2.2.10 of the Linux kernel.
- Dispatcher: The dispatcher was configured the same as the servers.
- Infrastructure: The clients all used ZNYX 346 100 Mbps Ethernet cards. The servers and the dispatcher all used Intel EtherExpress Pro/100 interfaces. All systems had a dedicated switch port on a Cisco 2900 XL Ethernet switch.
- Software: The servers ran version 1.3.6 of the Apache web server [4] while the clients ran `httperf` [22], a configurable HTTP load generator from Hewlett-Packard.

4.2. Httperf

`Httperf` [22] is a configurable HTTP load generator from Hewlett-Packard. While WebStone is also a very popular tool for web server benchmarking, we feel that `httperf` provides some additional features that WebStone does not. Most notably, we feel that `httperf` employs a more realistic model of user behavior. WebStone relies exclusively on operating system facilities to determine connection timeout, retries, etc. In contrast, `httperf` provides the user the ability to set a timeout. Just as a real user would, in the event that the web server has not responded within a reasonable amount of time (2 seconds in our tests), `httperf` will abort the connection and retry.

Additionally, WebStone attempts to connect to the web server as quickly as possible. `Httperf` on the other hand allows the user to select the connection rate manually. This provides the ability to examine the effect of increasing load on the web server in a controlled fashion.

Finally, the duration of WebStone tests are less controlled than `httperf`'s. As the deadline for the test expires, WebStone stops issuing new requests. However, outstanding requests are allowed to complete. With no timeout, this may be several minutes on some machines. `Httperf` terminates the test as soon as the deadline expires.

4.3. Results

Our results demonstrate that in tests of real-world (and some not-so-real-world) scenarios, our SASHA architecture provides a high level of fault tolerance. In some cases, faults might go unnoticed by users since they are detected and masked before they make a significant impact on the level of service. As expected, a dispatcher fault has the greatest impact on performance during fault detection and recovery. In the worst case, it took almost 6 seconds to detect and fully recover from a dispatcher fault; in the best case, it took less than 1.5 seconds.

Our fault-tolerance experiments are structured around three levels of service requested by client browsers: 2500 connections per second (cps), 1500 cps, and 500 cps. At each requested level of service, we measured performance for the following fault scenarios: no-faults, a dispatcher fault, one server fault, two server faults, three server faults, and four server faults. Figure 4 summarizes the actual level of service provided *during the fault detection and recovery interval* for each of the failure modes. In each fault scenario, the final level of service was higher than the level of service provided during the detection and recovery process. The rest of this section details these experiments as well as the final level of service provided after fault recovery.

4.3.1. 2,500 Connections Per Second

In the first case, we examined the behavior of a cluster consisting of five server nodes and the K6-2 400 dispatcher. Each of our five clients generated 500 requests per second. This was greater than the maximum sustainable load for our servers, though other tests have shown that a K6-2 400 dispatcher is capable of supporting over 3,300 connections per second. Each test ran for a total of 30 seconds. This short duration allows us to more easily discern the effects of node failure. Figure 4 shows that in the base, non-faulty, case the cluster is capable of servicing 2,465 connections per second.

In the first fault scenario, the dispatcher node was unplugged from the network shortly after beginning the test. We see that the average connection rate drops to 1,755 connections per second (cps) during the fault detection and recovery interval. This is to be expected, given the time taken to purge the ring and detect the dispatcher's absence. Following the startup of a new dispatcher, throughput returned to 2,000 cps, or $\frac{4}{5}$ of the original rate. Again, this is not surprising as the servers were operating at capacity previously and thus losing one of five nodes drops the performance to 80% of its previous level.

Next we tested a single-fault scenario. In this case, shortly after starting the test, we removed a server from the network. Results were slightly better than expected. Factoring in the connections allocated to the server before its loss was detected and given the degraded state of the system following diagnosis, we still managed to average 2,053 connections per second.

In the next scenario, we examined the impact of coincident faults. The test was allowed to get underway and then one server was taken off line. As the system was detecting this fault, the next server was taken off line. Again, we see a nearly linear performance decrease in performance as the connection rate drops to 1,691 cps.

The three fault scenario was similar to the two fault scenario, save that performance ends up being 1,574 cps. This relatively high performance–given that there are, at the end of the test, only two active servers–is most likely due to the fact that the state of the server gradually degrades over the course of the test. We see similar behavior with a four fault scenario. By the end of the four fault test, performance had stabilized at just under 500 cps, the maximum sustainable load for a single server.

4.3.2. 1,500 Connections Per Second

This test was similar to the 2,500 cps test, but with the servers less utilized. This allows us to observe the behavior of the system in fault-scenarios where we have excess server capacity. In this configuration, the base, no-fault, case shows 1,488 cps. As we have seen above, the servers are capable of servicing a total of 2,465 cps, therefore the cluster is only 60% utilized.

Similar to the 2,500 cps test, we first removed the dispatcher midway through the test. Again performance drops, as expected–to 1,297 cps in this case. However, owing to the excess capacity in the clustered server, by the end of the test, performance had returned to 1,500 cps. For this reason, the loss and election of the dispatcher seems less severe, relatively speaking, in the 1,500 cps test than in the 2,500 cps test.

In the next test, a server node was taken off line shortly after starting the test. We see that the dispatcher rapidly detects and masks this. Total throughput ended up at 1,451 cps. The loss of the server was nearly undetectable.

Next, we removed two servers from the network, similar to the two-fault scenario in the 2,500 cps environment. This makes the system into a three-node server operating at full capacity. Consequently, it has more difficulty restoring full performance after diagnosis. The average connection rate comes out at 1,221 cps.

In the three fault scenario, similar to our previous three fault scenario, we now examine the case where the servers are overloaded after diagnosis and recovery. This is reflected in the final rate of 1,081 cps. Again, while the four fault case has relatively high average performance, by the end of the test, it was stable at a just under 500 cps, our maximum throughput for one server.

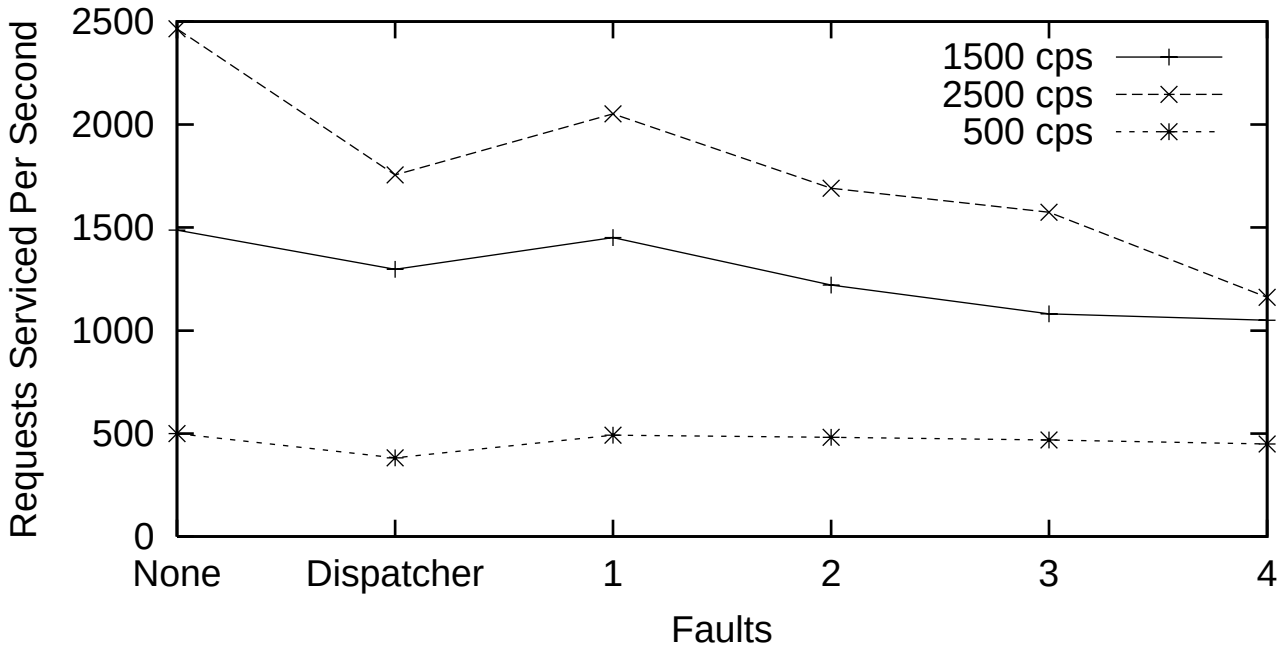

Figure 4. System performance, in requests serviced per second, during fault detection and recovery for three levels of requested service: 2500 connections per second (cps), 1500 cps, and 500 cps.

4.3.3. 500 Connections Per Second

Following the 2,500 and 1,500 cps tests, we examined a 500 cps environment. This gave us the opportunity to examine a highly under utilized system. In fact, we had an "extra" four servers in this configuration since one server alone is capable of servicing a 500 cps load.

This fact is reflected in all the fault scenarios. The most severe fault occurred with the dispatcher. In that case, we lost 2,941 connections to timeouts. However, after diagnosing the failure and electing a new dispatcher, throughput returned to a full 500 cps.

In the one, two, three, and four server-fault scenarios, the failure of the server nodes is nearly impossible to see on the graph. The final average throughput was 492.1, 482.2, 468.2, and 448.9 cps as compared with a base case of 499.4. That is, the loss of four out of five nodes over the course of thirty seconds caused a mere 10% reduction in performance.

5. Conclusion

There is a need for high performance web clustering solutions that allow the service provider to utilize standard server configurations. Traditionally, these have been based on custom operating systems and/or specialized hardware. While such solutions provide excellent performance, we have shown that our Scalable, Application-Space, Highly-Available (SASHA) architecture provides arbitrary levels of fault tolerance and performance sufficient for the most demanding environments. Moreover, the use of COTS systems throughout the cluster allows us to take advantage of the price/performance ratio offered by COTS systems while incrementally increasing the performance and availability of the server.

Our SASHA network-clustered server architecture consists of

- an application-space dispatcher, which performs layer 4 switching using layer 2 or layer 3 address translation;
- agent software that executes (in application space) on the server nodes to provide the capability for any server node to operate as the dispatcher;
- a novel distributed state-reconstruction algorithm, instead of the more typical state-replication approach for fault recovery; and
- a token-based communications protocol, TokenBeat, that supports self-configuring, detecting and adapting to the addition or removal of servers.

The SASHA architecture of clustering supports services other than web services with little or no changes to the application-space software developed for our prototype web server. It offers a flexible and cost-effective alternative to kernel-space or hardware-based solutions.

References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in web content hosting. In *1998 Workshop on Internet Server Performance*, June 1998.

[2] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second Intl. Conference on Software Engineering*, pages 562–570, 1976.

[3] E. Anderson, D. Patterson, and E. Brewer. The Magicrouter, an Application of Fast Packet Interposing. Submitted for publication in the Second Symposium on Operating Systems Design and Implementation, 17 May 1996.

[4] Apache Software Foundation. Apache Web Server. http://www.apache.org/.

[5] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for p-http in cluster-based web servers. In *Proceedings of 1999 USENIX Annual Technical Conference*, pages 185–198, June 1999.

[6] X. Chen and P. Mohapatra. Providing differentiated levels of services from a Internet server. In *Proceedings of IC3N'99: Eighth International Conference on Computer Communications and Networks*, pages 214–217, Oct. 1999.

[7] Cisco Systems Inc. Cisco 400 Series - LocalDirector. http://www.cisco.com/univercd/cc/td/doc/pcat/ld.htm, Apr. 2001.

[8] Cisco Systems Inc. Cisco CSS 1100. http://www.cisco.com/warp/public/cc/pd/si/11000/, Apr. 2001.

[9] M. E. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *Proceedings of the USITS*, 1999.

[10] Daemon9. Libnet: Network Routing Library, Aug. 1999. http://www.packetfactory.net/libnet/.

[11] O. Damani, P. Chung, Y. Huang, C. Kitala, and Y. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. In *Proceedings of the Sixth International WWW Conference*, Apr. 1997.

[12] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP-16)*, Oct. 1997.

[13] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. *Operating Systems Review*, 31(5):259–269, 1997.

[14] N. Fredrickson and N. Lynch. Electing a leader in a synchronous ring. *Journal of the ACM*, 34:98–115, Jan. 1984.

[15] X. Gan, T. Schroeder, S. Goddard, and B. Ramamurthy. LSMAC and LSNAT: Two approaches for cluster-based scalable web servers. In *ICC 2000*, June 2000.

[16] X. Gan, T. Schroeder, S. Goddard, and B. Ramamurthy. LSMAC vs. LSNAT: Scalable cluster-based web servers. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 3(3):175–185, 2000.

[17] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. on Computers*, 31:48–59, Jan. 1982.

[18] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. *Computer Networks and ISDN Systems*, Sept. 1999.

[19] IEEE. *Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language]*, 1996.

[20] Lawrence Berkeley Laboratory. Capture Library. ftp://ftp.ee.lbl.gov/libcap.tar.Z.

[21] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias. Design and Performance of a Web Server Accelerator. In *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, 1999.

[22] D. Mosberger and T. Jin. httperf–A Tool for Measuring Web Server Performance. ftp://ftp.hpl.hp.com/pub/httperf/.

[23] E. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: a Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39, 1996.

[24] Nortel Networks. Alteon ACEdirector. http://www.alteonwebsystems.com/products/acedirector/, Apr. 2001.

[25] Nortel Networks. Alteon Personal Content Director (PCD). http://www.alteonwebsystems.com/products/PCD/, Apr. 2001.

[26] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceeding of the ACM Eighth International Conference on Architectural Support for Progamming Languages and Operating Systems (ASPLOS-VIII)*, Oct. 1998.

[27] R. Pandey, J. Barnes, and R. Olson. Supporting quality of service in http servers. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 247–256, June 1998.

[28] G. Rao. Application Level Differentiated Services for Web Servers. Technical report, Dept. of Computer Science & Engineering, University of Nebraska-Lincoln, Apr. 2000.

[29] F. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, 1984.

[30] T. Schroeder and S. Goddard. The tokenbeat protocol. Technical Report UNL-CSCE-99-526, Dept. of Computer Science & Engineering, University of Nebraska-Lincoln, Dec. 1999.

[31] T. Schroeder, S. Goddard, and B. Ramamurthy. Scalable web server clustering technologies. *IEEE Network*, 14(3):38–45, May/June 2000.

[32] S. Singh and J. Kurose. Electing 'good' leaders. *Journal of Parallel and Distributed Computing*, 21:184–201, May 1994.

[33] P. Srisuresh and D. Gan. *Load Sharing Using Network Address Translation*. RFC 2391, The Internet Society, Aug. 1998.

[34] R. van Renesse, K. Birman, and S. Maffeis. Horus, a Flexible Group Communication System. *Communications of the ACM*, 39, 1996.

[35] C. Wei. A QoS assurance mechanism for cluster-based web servers. Technical report, Dept. of Computer Science & Engineering, University of Nebraska-Lincoln, Dec. 2000.

[36] Zeus Technology Ltd. Zeus Technology. http://www.zeus.co.uk/, Apr. 2001.