Socket Cloning for Cluster-Based Web Servers*

Yiu-Fai Sit, Cho-Li Wang, Francis Lau

Department of Computer Science and Information Systems
The University of Hong Kong
E-mail: {yfsit, clwang, fcmlau}@csis.hku.hk

Abstract

Cluster-based web server is a popular solution to meet the demand of the ever-growing web traffic. However, existing approaches suffer from several limitations to achieve this. Dispatcher-based systems either can achieve only coarse-grained load balancing or would introduce heavy load to the dispatcher. Mechanisms like cooperative caching consume much network resources when transferring large cache objects. In this paper, we present a new network support mechanism, called Socket Cloning (SC), in which an opened socket can be migrated efficiently between cluster nodes. With SC, the processing of HTTP requests can be moved to the node that has a cached copy of the requested document, thus bypassing any object transfer between peer servers. A prototype has been implemented and tests show that SC incurs less overhead than all the mentioned approaches. In trace-driven benchmark tests, our system outperforms these approaches by more than 30% with a cluster of twelve web server nodes.

1. Introduction

Cluster-based web servers have become a common solution to high-traffic web hosting in recent years. Such a system usually consists of a dispatcher and a collection of web server nodes connected to a local area network. Client requests are distributed to the web server nodes by the dispatcher to obtain high throughput. Existing solutions for improving cluster-based web server performance either follow a dispatcher-based approach to achieve load balancing or use some special caching mechanisms that can avoid excessive disk operations.

The dispatcher-based approach emphasizes on request distribution to achieve load balancing. All the packets from the clients to the cluster will first reach the dispatcher, which then routes the packets to the web server nodes based on some load balancing policies. There exist many methods to implement such a request distribution, which can be divided into two categories based on the OSI layer at which the distribution decision is made [10, 11].

In layer-4 dispatching, e.g., Magicrouter [2] and Network Dispatcher [21], the decision is made according to the IP address and TCP port number of the client. The dispatcher routes packets to the chosen node, which establishes TCP connection directly with the client. Since the endpoints in a TCP connection cannot be changed, the dispatcher has to route the packets of the same connection to the same web server node. Fine-grained load balancing is therefore difficult to achieve. The dispatcher cannot change the assigned node of an established connection even when other nodes in the cluster become idle. In addition, the centralized dispatcher is a single point of failure to the whole cluster. A solution has been proposed in Distributed Packet Rewriting [8, 9] that allows several distributed dispatchers to be present in a cluster. While this approach removes the single point of failure, it still has the other drawbacks of layer-4 dispatching.

Layer-7 dispatching, often referred to as content-based distribution, looks into a request to decide which cluster node should serve the request. Since HTTP request is sent after the TCP connection is established, the dispatcher has to first establish a connection with the client before any distribution decision can be made. After that, the processing of the request is passed to the chosen web server node. *TCP handoff* [6, 7, 13, 25, 27, 31] and *TCP splicing* [15, 28] are two mechanisms that support such operation. In TCP handoff, the connection endpoint in the dispatcher is passed to the chosen node, which then processes the request and sends out the response directly to the client. In TCP slicing, the dispatcher acts like a proxy. The dispatcher receives a request from the client and then forwards it to the chosen web server node. The HTTP response is sent back to the dispatcher, which in turn forwards it to the client. This limits the throughput of the whole web server because all the traffic has to pass through the dispatcher. Figure 1 and 2 show the logical flow of how a series of HTTP requests are

*This research is supported by HKU's seed funding programmes for basic and applied research under account code 10203944.

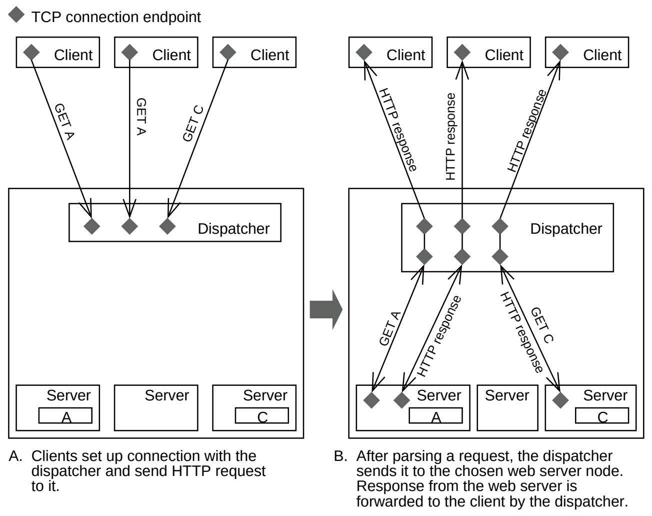
TCP connection endpoint
Client
Client
Client
GET A
GET A
GET C
Dispatcher
Server
Server
Server
A
C
HTTP response
GET A
GET C
A. Clients set up connection with the dispatcher and send HTTP request to it.
B. After parsing a request, the dispatcher sends it to the chosen web server node. Response from the web server is forwarded to the client by the dispatcher.

Figure 1. TCP Splicing

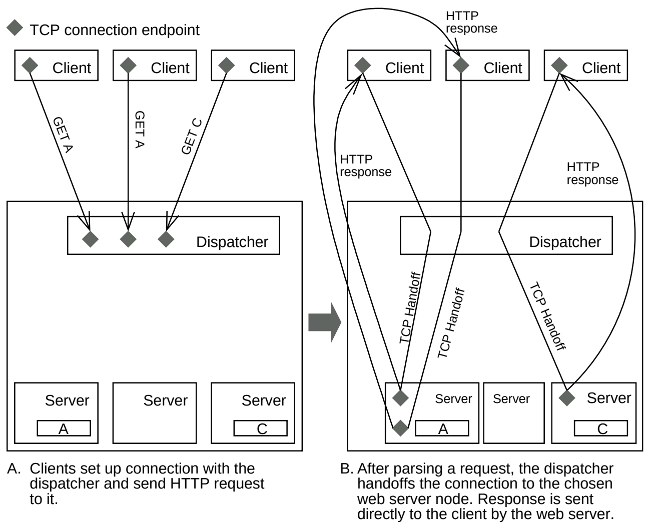
TCP connection endpoint
Client
Client
Client
GET A
GET A
GET C
Dispatcher
Server
Server
Server
A
C
HTTP response
TCP Handoff
A. Clients set up connection with the dispatcher and send HTTP request to it.
B. After parsing a request, the dispatcher handoffs the connection to the chosen web server node. Response is sent directly to the client by the web server.

Figure 2. TCP Handoff

processed in a cluster-based web server with TCP splicing and TCP handoff respectively.

Since the handoff or splicing module has to parse every request in order to make the distribution decision, every request is actually parsed twice: once in the module and once in the web server. Establishing a TCP connection with each client and parsing the HTTP request introduce much overhead, and the dispatcher can become a performance bottleneck much more easily than the layer-4 approach. Although a solution has been proposed in LARD [7] to share the dispatcher's load among the web server nodes, this in turn imposes more work in each of them. The overall performance is thus limited.

Supporting persistent HTTP is another challenge in TCP handoff. The main issue is to prevent the TCP stream from draining during the process of handoff. This problem is mentioned in [6] but there is no solution given. The lack of such mechanism limits TCP handoff's advantage over layer-4 dispatching in persistent HTTP connections because only the first request in a connection can be dispatched according to its content. Subsequent requests in the same connection are still bound to the same server node, as in layer-4 dispatching. Layer-7 approaches are also inefficient for small-size replies where the dispatching overhead can be much larger than the actual transfer time of the replies.

In addition to the dispatcher-based approach, the idea of cooperative caching [17] has also been relied upon for achieving high performance [1, 12, 16, 31]. Each web server node maintains a cache pool for web documents. When a node does not have the web document requested by the client in its cache, it will first try to fetch the object in another node's memory cache instead of reading it from local disk. A mechanism called cache forwarding (*CF*) is thus needed to transfer cached objects between the nodes. This cache transfer between peer servers is the major cost in cooperative caching and works best only with small-size transfers. For larger ones, reading from the local hard disk is probably more efficient as the disk usually has a higher transfer rate.

Most of the commercial products for cluster-based web servers follow the dispatcher approach [14, 18, 19, 24]. A web switch or a load balancer sits in front of the cluster and distributes the requests to the web server nodes by layer-4 or layer-7 policies. Mechanisms similar to TCP splicing are usually adopted in these switches to achieve layer-7 load balancing. We are only aware of an exception from Resonate, Inc. [26], which offers a mechanism that is similar to TCP handoff. All these commercial products share the same advantages and drawbacks of dispatcher-based approaches, however.

In this paper, we propose a new network support mechanism for cluster-based web servers, called Socket Cloning (*SC*), which aims at resolving the aforementioned problems and providing a general solution for cluster-based service to achieve high performance. Our approach allows web server software to move opened sockets efficiently between cluster nodes. A request can therefore be processed in the web server node with the document in its cache to achieve load balancing and high performance without the need to transfer the cached copy.

The organization of this paper is as follows. In Section 2, we present the architecture of Socket Cloning. The design of a cluster-based web server with Socket Cloning is discussed in Section 3. Performance tests and evaluations of Socket Cloning and different cluster-based web servers are presented in Section 4. We conclude in Section 5.

2. Socket Cloning

In this section, we describe in detail the architecture and protocol of Socket Cloning.

2.1. System Architecture

Our proposed architecture of Socket Cloning aims at providing an efficient and transparent network support system for cluster-based web servers with the discussed problems resolved. For the simplicity of the discussion, it is assumed that a layer-4/2 dispatcher is used to distribute requests to the web server nodes in the cluster. Indeed, any other lightweight dispatching mechanism can also be used. To avoid the dispatcher becoming the single point of failure, techniques like Distributed Packet Rewriting [8, 9] can be used as well. There are three components in Socket Cloning: *SC Client*, *SC Server*, and *Packet Router*. The architecture is shown in Figure 3.

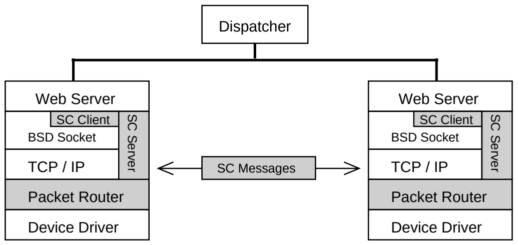

Figure 3. System Architecture of SC

In this system architecture, SC Client provides a system call interface to the web server in the node. When a web server decides to make another node to handle the request, it issues the system call provided by SC Client to clone the socket. SC Client then packs all the relevant information of the opened socket and sends it out to the SC Server in the remote node through a persistent connection. The whole message is called *SC Message*. When the cloning system call returns, the web server treats the request as served and processes the next request.

When SC Server receives an SC Message, it will create a socket called *clone*. The states of the clone and the protocol stack are then reconstructed according to the information in the SC Message. After that, the clone is native to this node and subsequent packets will go through its normal network protocol stack. There is no extra overhead in processing the packets of a clone. Outgoing packets of the clone are sent directly to the client. Upon successful cloning, the SC Server will send an acknowledgement back to the SC Client. It will then inform the Packet Router to route subsequent packets for that socket to the clone's node. During the execution, packets from the client will first reach the original node and be routed to the clone's node while packets to the client are sent directly from the clone's node. A triangular routing path is established. Furthermore, packets that contain non-zero TCP payload are passed to the network stack of the original node as well as routed to the clone.

After cloning, the original socket remains in its node. It will not be destroyed until the connection is closed. In persistent HTTP, the original socket will handle further messages received in the connection after cloning. The clone will be closed after serving a request. Figure 4 shows how a series of HTTP requests are handled in the system.

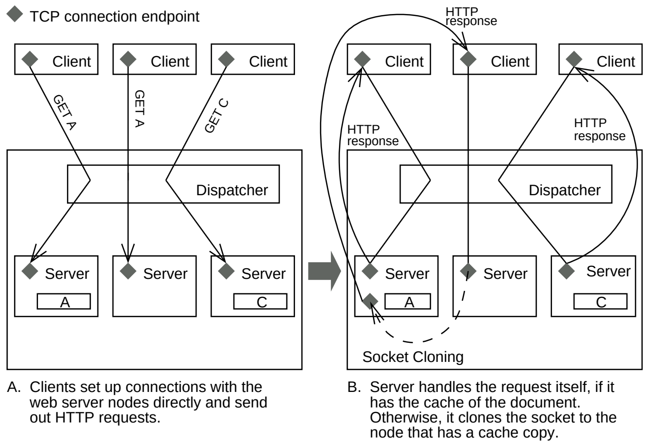
TCP connection endpoint
Client
Client
Client
GET A
GET A
GET C
Dispatcher
Server
A
Server
Server
C
A. Clients set up connections with the web server nodes directly and send out HTTP requests.
HTTP response
Client
Client
Client
HTTP response
HTTP response
Dispatcher
Server
A
Server
Server
C
Socket Cloning
B. Server handles the request itself, if it has the cache of the document. Otherwise, it clones the socket to the node that has a cache copy.

Figure 4. SC in Cluster-Based Web Server

2.2. Mechanism

When a web server in a cluster node decides not to handle a request, it clones the socket of this request to another node by calling the SC Client to send out an SC Message, which has four fields. The first field defines the total length of the message. The second field contains the states of the socket and protocol stack such as windows sizes. The length of this field is fixed. The third field is the socket seed, which contains the IP and TCP headers that the server receives from the client at the time of cloning. It represents the basic information of the IP and TCP stack. The last field is the Application State Field (*ASF*). It represents the request processing state and is generated by the cloning web server when the system call is invoked. For example, the ASF can be the client's request with the RANGE header added to specify how many bytes have been successfully transferred. The second field and the ASF essentially form an IP packet containing HTTP request. The first three fields are composed by the SC Client and the ASF is supplied by the calling web server.

After an SC Message is received, the SC Server will create a socket, i.e. the clone, using the socket seed. Since the socket seed is essentially an IP packet, basic information of the original socket's IP and TCP stack can be reconstructed by treating the socket seed as a TCP SYN packet to set up the clone. To do this, the sequence number is reduced by one and the SYN bit is turned on in the socket seed. After that, a fake three-way handshaking is done by the SC Server to setup the clone using the modified socket seed. As the sequence number is increased by one after the handshaking, the sequence number becomes the same as before. The sequence number of the clone itself is also set up to match the one in the socket seed. Other states of the socket such as windows sizes, maximum segment size, and timestamp values are reconstructed using the second field of the SC Message. The ASF will be put in the clone's receive buffer, which becomes the client's request to the web server. The clone is now set up and is treated as an ordinary socket in this node.

2.3. Triangular Implicit State Synchronization

The clone and the original socket have the same state right after cloning. However, the states of the clone and the original socket can become different when the clone has sent out some packets. As the original node will process further requests from the client, the states of the original and cloned socket have to be synchronized. Packets sent from an unsynchronized socket will appear erratic to the client and packet from the client will be treated as an abnormal one in such socket. The states of the sockets thus have to be consistent before the original socket can take over control of the connection again to serve subsequent requests in a connection.

Because of triangular routing, the original socket does not know what the clone has sent. As a result, some mechanism is needed to update the states of the original socket to keep the states consistent. This usually involves internode communications and may adversely affect scalability of the whole system. In our system, explicit synchronization is not required. The Packet Router takes full advantage of the acknowledgement received from the client to manipulate the state of the original socket so that the states become synchronized after the acknowledgment is received at the clone's node. Such implicit synchronization removes all the inter-node communication for state information exchange. It allows an efficient and scalable design for multiple-cloning without the need for explicit control, as we shall see in Section 3.

Figure 5 shows an example on how the synchronization is carried out. At the beginning (0), the original and the cloned socket have the same state (seq = nxt = 100). The state of the clone changes (1) when it sends out a message to the client (2). After the message is received by the client (3), it will send out an acknowledgement (4). The Packet Router, knowing the socket has been cloned, will look into

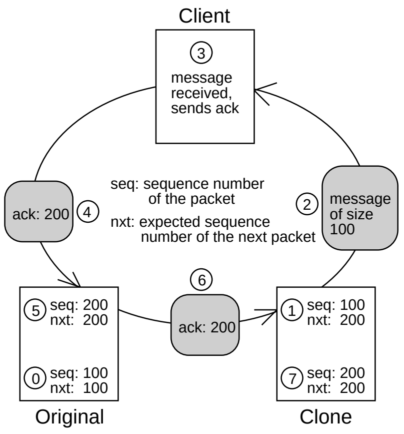

Figure 5. Triangular Implicit State Synchronization

the contents of the packet and update the socket's state accordingly (5). This step is done right above the device driver level and does not involve the operation of the normal network protocol stack. The acknowledgement is routed to the clone's node (6) by the Packet Router. This packet traverses the normal network stack in the clone's node and the states are updated (7). In this way, the states of the original and cloned socket are synchronized without explicit communications between the two cluster nodes. This reduces the overhead and improves the scalability of the system.

The TCP state transition and packet flow of a closing connection depends on which endpoint sends the FIN packet first. Since the original node cannot see the outgoing packets of the clone, it cannot differentiate whether it is the clone or the client that does the active close. The Packet Router thus does not know how many packets it should expect to route for the connection when a FIN packet is received. This problem is solved by setting a timer when such packet is seen. The Packet Router stops to route packets of the connection when its timer expires. This again eliminates the need for explicit synchronization.

3. Cluster-Based Web Server with SC

This section describes how the web server interacts with the Socket Cloning system. We also introduce our prototype implementation.

3.1. SC in Cluster-Based Web Server

The whole operation of a cluster-based web server using Socket Cloning is as follows. The web server software in a cluster node first establishes connection with clients directly and parses the request. A mapping function is then

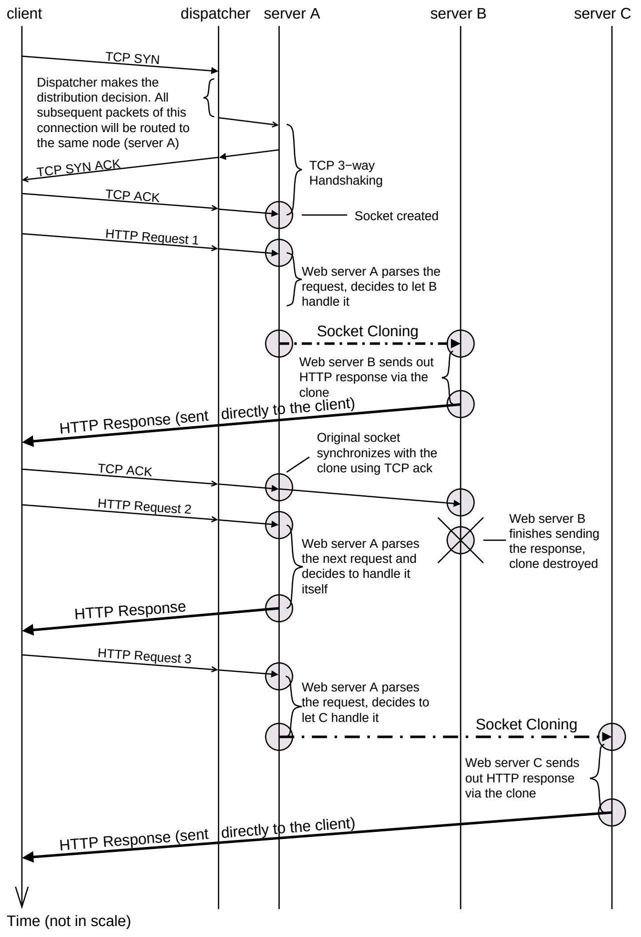
client
dispatcher
server A
server B
server C
TCP SYN
Dispatcher makes the distribution decision. All subsequent packets of this connection will be routed to the same node (server A)
TCP SYN ACK
TCP 3−way Handshaking
TCP ACK
Socket created
HTTP Request 1
Web server A parses the request, decides to let B handle it
Socket Cloning
Web server B sends out HTTP response via the clone
HTTP Response (sent directly to the client)
Original socket synchronizes with the clone using TCP ack
TCP ACK
HTTP Request 2
Web server B finishes sending the response, clone destroyed
Web server A parses the next request and decides to handle it itself
HTTP Response
HTTP Request 3
Web server A parses the request, decides to let C handle it
Socket Cloning
Web server C sends out HTTP response via the clone
HTTP Response (sent directly to the client)
Time (not in scale)

Figure 6. Workflow of a Cluster-Based Web Server with SC

applied to the request to decide which node should process the request. Various mapping functions using load-based, location-based, cache-based, or other algorithms could be adopted. The web server will handle the request directly if it is the one that the function maps to. Otherwise, it will clone the socket to the chosen node and let it serve the request.

For persistent HTTP, we provide an efficient and scalable mechanism for cloning the socket multiple times so that every request is served by the most appropriate node. The requests in a persistent HTTP connection are processed one after another. If it is a clone that handles the request, the web server in the clone's node will ignore the subsequent requests in the connection and close the clone after sending out the response. The web server in the original node then handles the next request. It can either clone the socket again to let another node to handle the next request, or it can handle the request itself. This process continues until all the requests within a connection have been handled. In this way, multiple-cloning is supported without the need to synchronize the web server nodes or to prevent the TCP stream from draining during cloning. Figure 6 summarizes the workflow of a cluster-based web server with Socket Cloning for a non-pipelined persistent HTTP connection.

Some modern web browsers support pipelining of requests [23] in which a browser can send out requests before receiving a complete response is also supported. SC is also designed to handle this service. The length of the requested file is noted by the original node when it clones the socket. The Packet Router compares the acknowledged sequence number from the client and the pre-computed message size. When the whole response has been acknowledged by the client, the web server in the original node is informed by the Packet Router to process the next request. As the states of the clone and original sockets are synchronized automatically, the original node can take over control of the connection right after the clone's node has finished serving the request. This ensures all the requests within a connection are served properly and efficiently without explicit control of the nodes.

There are several advantages of Socket Cloning in cluster-based web servers: 1.) The cloned socket sends packets directly to the client and does not need to be forwarded to the original socket first as in the TCP splicing and some layer-4 approaches. 2.) With SC, the processing of HTTP requests can be migrated to the node that has a cached copy of the requested document, thus bypassing any cache transfer between cluster nodes. 3.) Once cloned, the socket becomes native to the node and no extra network processing layer or wrapper is needed. 4.) Multiple-cloning of the same socket is also allowed in SC, which makes it possible to achieve fine-grained load balancing in persistent HTTP. 5.) SC also has much lower overheads and thus higher scalability than the other approaches, as we shall see in the next section.

3.2. Prototype Implementation

We have implemented a prototype of Socket Cloning in Linux, kernel version 2.4.2. The network stack has been modified so that a clone can be set up without a real connection. Normal network operations are not affected by the change. The SC Server, SC Client, and the Packet Router are all implemented as kernel modules. These modules have to be loaded in all the cluster nodes before any web server can clone a socket. When the system starts, SC Client connects to the SC Servers in the other nodes of the cluster. All the SC Messages are sent through these connections without the need to start a new one for each message. We have also modified kHTTPd [22], a kernel-based web server, to make use of the SC facilities.

4. Benchmarks Results

We have carried out two sets of benchmarks. In micro-benchmark, we compare the performance of different content-aware mechanisms. Throughputs of cluster-based web servers employing different techniques are evaluated in trace-driven benchmark.

4.1. Micro-Benchmark

To evaluate the performance of Socket Cloning, we have carried out tests with three mechanisms that are content-aware: SC, cooperative caching, and LARD [25]. The cooperative caching scheme is implemented in kernel space in Linux for comparison because both LARD and SC run in kernel space. The LARD source code is obtained from the project's homepage and runs in FreeBSD 2.2.6. Tests were carried out in two clusters: an Intel Pentium II 300MHz cluster and an Intel Pentium III 733MHz cluster. All the nodes are connected via a Fast Ethernet switch.

In these tests we used ApacheBench [3], a simple benchmark from Apache HTTP Server's source that retrieves a single document repeatedly. The tested mechanism was applied to every request. For example, in the test with cooperative caching, cache forwarding (CF) is always performed. Figure 7 illustrates how these tests were carried out.

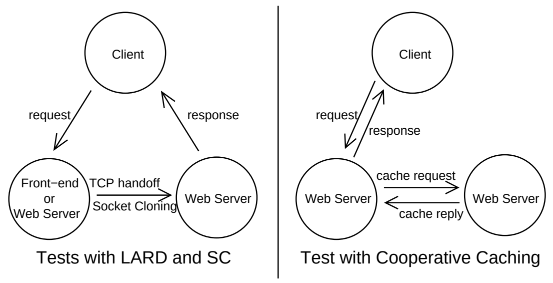

Figure 7. Micro-Benchmark Test

We found that Socket Cloning has the least relative startup cost. The total latency in sending the SC Message and setting up the clone is measured to be 134 μs in a Pentium II 300 node and 46 μs in a Pentium III 733 node, whereas it is 194 μs for Pentium II 300 machines in LARD. One may argue that these numbers depend on the operating system and do not represent the true cost. To minimize the effects of the different underlying environments and compare the relative overhead of the three mechanisms, a metric called *normalized efficiency* is used. To compute the normalized efficiency, throughput of the tested mechanism is first obtained. It is then divided by the throughput of a single web server running in the same environment. Figure 8 shows the result.

When the requested file is small, the startup cost of a mechanism is the main contributor of the decreased efficiency. The smaller normalized efficiency achieved in LARD shows that it have larger setup cost than the other two mechanisms.

In cooperative caching, the major cost is in cache forwarding. There are two messages involved in each CF: one is the request for a cache copy and the other is the cache reply. The normalized efficiency stays about the same for files of less than one kilobyte. It gradually drops to about 50% when larger files are transferred because the time to forward the cache becomes more significant. This shows the drawback of cache forwarding. The requested file is actually sent twice, once in cache forwarding and once in the reply. The crossover point of CF and LARD is about 3 kilobyte, which is similar to the result in [6].

In both of LARD and SC, there is a triangular routing path between the client and the request-handling web server node. Client packets have to be routed by an intermediate forwarding node in the cluster to reach the web server node while response packets are sent directly to the client. The overhead in forwarding client acknowledgements thus becomes more significant in large response. In LARD, the IP address and TCP port number in the client packets have to be modified in both of the front-end and the handed off node. This adds extra delays in every client packet. As a result, it can only achieve less than 80% normalized efficiency. In SC, the Packet Router acts as a layer-4 dispatcher and the clone's node does not have to modify the received packets. This scheme has lower overhead and SC is able to achieve nearly 100% normalized efficiency for large files.

Note that in real systems, SC or CF does not have to be carried out in all requests. If the node that the web client connects to has a cached copy of the document, then SC or CF is not necessary. Such situation can be increased if there are replications of cache copies in the cluster. On the other hand, the cost of TCP handoff has to apply to every connection in LARD because all the clients must establish connection with the front-end.

4.2. Trace-Driven Benchmark

We have carried out trace-driven benchmark tests with cooperative caching (CF) and SC by replaying a log file. LVS [29] with round-robin request distribution using direct routing is also tested for comparison. It is also used to distribute requests to the web server nodes in CF and SC. Two cache replacement policies, LRU and LFU, are used in cooperative caching. The mapping function used in SC is a hash function of the HTTP request. In all the tests, kHTTPd [22] is run in each of the server nodes.

The log file we used is collected by the Computer Science Division, EECS Department, of UC Berkeley [30]. It

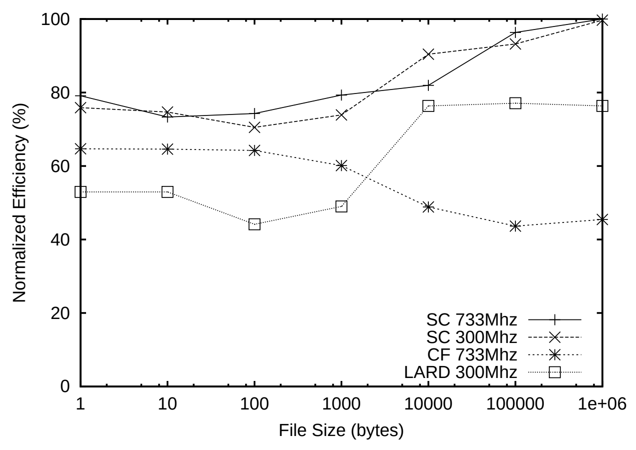

Figure 8. Normalized Efficiency

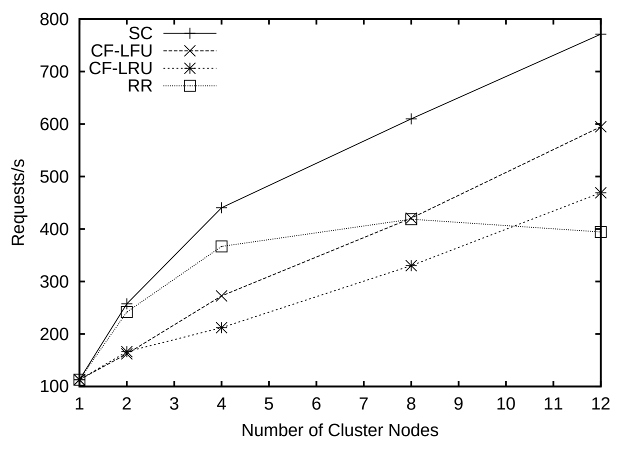

Figure 9. Trace-Driven Benchmark

recorded all the web accesses of the Division's website in January 2001. The extracted log file has 3940707 requests. 87425 unique files are referenced and they span a total size of about 6.6 GB. The average size of the requested document is 40KB.

Our testing environment consists of 1 to 12 web servers and 19 clients. All the machines are connected to a single Fast-Ethernet switch. All the web server nodes have a 733MHz Intel Pentium III CPU and 384MB of main memory. Each client node runs http_load [20] with 2 concurrent connections. Http_load is modified to replay the log in order. All the files are stored in a 6-way SMP server connected in the same LAN and all the web servers mount the web documents by NFS. Figure 9 shows the requests throughput of the tests.

In these tests, the total size of the documents is much larger than the sum of the memories in all the cluster nodes. The performance of a web server thus depends on how efficient the memories of the nodes are used to avoid disk access. The throughput of RR stops to increase when more than eight server nodes are used. This is because layer-4 dispatching cannot distribute requests to the web server nodes according to its content. The cache in each of the server nodes thus is responsible to host the whole set of web document. This inefficient use of the memory in the cluster limits its scalability.

We found cooperative caching with an LFU cache replacement policy has better performance than the one that uses LRU, which is typical for web access patterns [12]. This is related to the very low temporal locality of the access pattern in the common characteristics of web server workload [4, 5]. However, cooperative caching does not have higher throughput than RR until more than eight web server nodes are used. This is mainly due to the overhead involved in large cache transfers. Since all the nodes have only one network interface, a node cannot forward a cache and send reply to the test client at the same time. The network resource is shared by cache transfers and HTTP responses. This limits the performance of the whole web server.

Although SC uses file system cache, which employs LRU replacement policy, it still outperforms cooperative caching with LFU and the other tested mechanisms. This is because of the lower overhead in SC. This result is encouraging and higher performance is expected to be achieved if other cache replacement policy such as LFU that suits the request pattern is used in SC. SC also has nearly twice the throughput of cooperative caching with LRU in most cases. This confirms our result in the micro-benchmark. With twelve web server nodes, SC outperforms CF-LFU, CF-LRU and RR by 30%, 64%, and 96% respectively.

5. Conclusions

In this paper, we have proposed a novel network support mechanism for cluster-based web servers, called Socket Cloning. Socket Cloning allows an opened socket to be moved efficiently between cluster nodes. It incurs less overhead than existing content-aware approaches, such as TCP handoff and caching forwarding. This enhances the performance of cluster-based web servers. Triangular implicit state synchronization is used in Socket Cloning so that unnecessary inter-node communications are not required. This results in a more scalable system. Multiple-cloning is also supported to provide fine-grained load balancing in persistent HTTP connections without synchronization and TCP stream drainage problems. We have also evaluated the performance of a cluster-based web server with Socket Cloning and compared it with the cooperative caching approach using different cache replacement policies by trace-driven benchmarks. Performance results indicate that our approach can outperform other mechanisms.

Although a cluster-based web server with Socket Cloning can outperform other similar systems, there are still some other optimizations left to be explored. For example, in persistent HTTP, the clone's web server can also parses the next request instead of closing the clone after serving a request. As a result, socket does not have to be cloned again if the next request is then determined to be handled by the clone's node. In addition, sockets can be pre-cloned in several nodes for processing the different requests within a pipelined persistent HTTP connection. Requests handling can thus be overlapped. When a web server has finished sending out the response, it can inform the clone in another node to serve the next request. The next response can thus be sent out immediately by that node without the latency in cloning the socket. While this can improve the performance, this will require more sophisticated protocols among the components in the Socket Cloning architecture.

References

[1] W. H. Ahn, S. H. Park, and D. Park. Efficient cooperative caching for file systems in cluster-based web servers. *International Conference on Cluster Computing*, Nov. 2000.

[2] E. Anderson, D. Patterson, and E. Brewer. The Magicrouter: An appilcation of fast packet interposing. *Second Symposium on Operating Systems Design and Implementation*, May 1996.

[3] Apachebench, http://www.cpan.org/modules/by-module/HTTPD/.

[4] M. F. Arlitt and T. Jin. A workload characterization of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, May/June 2000.

[5] M. F. Arlitt and C. L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, Oct. 1997.

[6] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for p-http in cluster-based web servers. In *Proceedings of the USENIX 1999 Annual Technical Conference*, jun 1999.

[7] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of the USENIX 2000 Annual Technical Conference*, June 2000.

[8] L. Aversa and A. Bestavros. Load balancing a cluster of web servers using distributed packet rewriting. Technical Report 1999-001, CS Department, Boston University, Jan. 6 1999. Thu, 13 Jun 2002 17:36:00 GMT.

[9] A. Bestavros, M. Crovella, J. Liu, and D. Martin. Distributed packet rewriting and its application to scalable server architectures. Technical Report 1998-003, CS Department, Boston University, Feb. 1 1998.

[10] H. Bryhni, E. Klovning, and O. Kure. A comparison of load balancing techniques for scalable web servers. *IEEE Network*, 14(4):58–64, July/Aug. 2000.

[11] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on Web-server systems. *IEEE Internet Computing*, 3(3):28–39, May/June 1999.

[12] G. Chen, C. L. Wang, and F. C. Lau. Building a scalable web server with global object space support on heterogeneous clusters. *International Conference on Cluster Computing*, Oct. 2001.

[13] L. Cherkasova and M. Karlsson. Web server cluster design with workload-aware request distribution strategy with ward. In *Proceedings of the 3rd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, pages 212–221, June 2001.

[14] Cisco Systens, Inc. Css 11100 content services switch, http://www.cisco.com/.

[15] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems (USITS-99)*, pages 117–126, Berkeley, CA, Oct. 11–14 1999.

[16] F. M. Cuenca-Acuna and T. D. Nguyen. Cooperative caching middleware for cluster-based servers. In *10th IEEE International Symposium on High Performance Distributed Computing*, Aug. 2001.

[17] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, Monterey, California, Nov. 1994.

[18] F5 Networks, Inc. Big-IP Controller, http://www.f5.com/.

[19] Foundry Networks, Inc. ServerIron, http://www.foundrynetworks.com/.

[20] http_load. http://www.acme.com/software/http_load/.

[21] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Network Dispatcher: A connection router for scalable Internet services. In *Proceedings of the 7th International World Wide Web Conference*, Apr. 1998.

[22] kHTTPd - Linux HTTP accelerator. http://www.fenrus.demon.nl/.

[23] J. C. Mogul. The case for persistent-connection HTTP. In *Proceedings of the SIGCOMM'95 conference*, Cambridge, MA, Aug. 1995.

[24] Nortel Networks Ltd. Alteon ACEdirector, http://www.nortelnetworks.com/.

[25] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. *ACM SIGPLAN Notices*, 33(11):205–216, Nov. 1998.

[26] Resonate, Inc. Central Dispatch, http://www.nortelnetworks.com/.

[27] W. Tang, L. Cherkasova, L. Russell, and M. W. Mutka. Modular TCP handoff design in STREAMS-based TCP/IP implementation. In *Proceedings fo the First International Conference on Networking*, pages 71–81, July 2001.

[28] TCP Splicing. http://www.linuxvirtualserver.org/software/tcpsp/index.html.

[29] The Linux Virtual Server Project. http://www.linuxvirtualserver.org/.

[30] University of California, Berkeley. CS Access Log Files, http://www.cs.berkeley.edu/logs/.

[31] Whizz Technology Ltd. EC-Cache Engine, http://www.whizztech.com/ec_cache.html.