SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:1215–1241
Published online 5 April 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.814

Efficient support for content-aware request distribution and persistent connection in Web clusters

Ho-Han Liu[1], Mei-Ling Chiang[2,*,†] and Men-Chao Wu[2]

[1]*Department of Computer Science, National Chiao-Tung University, HsinChu 330, Taiwan, Republic of China*
[2]*Department of Information Management, National Chi-Nan University, Puli, NanTou 545, Taiwan, Republic of China*

SUMMARY

To support Web clusters with efficient dispatching mechanisms and policies, we propose a light-weight TCP connection transfer mechanism, TCP Rebuilding, and use it to develop a content-aware request dispatching platform, LVS-CAD, in which the request dispatcher can extract and analyze the content in requests and then dispatch each request by its content or type of service requested. To efficiently support HTTP/1.1 persistent connection in Web clusters, request scheduling should be performed per request rather than per connection. Consequently, multiple TCP Rebuilding, as an extension to normal TCP Rebuilding, is proposed and implemented. On this platform, we also devise fast TCP module handshaking to process the handshaking between clients and the request dispatcher in the IP layer instead of in the TCP layer for faster response times. Furthermore, we also propose content-aware request distribution policies that consider cache locality and various types of costs for dispatching requests in this platform, which makes the resource utilization of Web servers more effective. Experimental results of a practical implementation on Linux show that the proposed system, mechanisms, and policies can effectively improve the performance of Web clusters. Copyright © 2007 John Wiley & Sons, Ltd.

Received 8 February 2005; Revised 3 October 2006; Accepted 20 December 2006

KEY WORDS: cluster-based systems; Web clusters; content-aware dispatching; persistent connection; Linux virtual server

*Correspondence to: Mei-Ling Chiang, Department of Information Management, National Chi-Nan University, Puli, NanTou 545, Taiwan, Republic of China.
†E-mail: joanna@ncnu.edu.tw

1. INTRODUCTION

The problems of how to deal with the rapid growth of traffic on Web sites and improve the quality of Web services are serious current challenges for enterprises involved in the construction of Web sites. A Web cluster is composed of a front-end request dispatching server, also called a Web switch, and several back-end request-handling servers. By distributing requests from clients to separate servers for load balancing or load sharing, Web clusters have proved to be a better solution than using an overloaded single server.

However, in recent years, as a consequence of the impact of e-Commerce and e-Business, the server-side dynamic page script, for example PHP, ASP, or JSP, has become increasingly popular, and the loads on Web servers are also much heavier. Owing to the large difference between dynamic pages and static pages in server resource consumption, Web clusters must be re-developed to suit the characteristics of dynamic pages instead of traditional static pages, especially in request distribution strategies for dispatching requests to the most appropriate request-handling servers.

Many studies [1–13] have focused on content-aware request distribution in which the content of request packets, i.e. URL information, is used as the basis for dispatching. Using the content of request and load information, complex or intelligent request dispatching algorithms, such as improving disk cache hit rates, can be applied. Thus, Web clusters will be more efficient in handling all types of Web pages. Furthermore, the partitioning of Web content, the construction of specialized Web services among Web servers, or maintaining session integrity can be achieved.

For Web clusters to perform content-aware request distribution, Web clusters must first establish a TCP connection with the client before acquiring the contents of request packets from clients. Only when the request packet has been received are the Web clusters able to proceed with request dispatching based on the information acquired. A connection transfer between the dispatching server and the request-handling server is therefore needed. However, the problem of how to make the response packets from the request-handling server correspond with the TCP connection that had been previously established by the client poses an immense challenge.

Some studies [1–5,8,9,11,14–16] have proposed TCP connection transfer mechanisms. However, most of them either require sending extra packets for connection transfer, which could result in the internal network of Web clusters being overloaded, or they intercept the inbound or outbound packets and modify the packets during the connection, which would affect the integral efficiency of the Web clusters. Therefore, we propose a light-weight TCP connection transfer mechanism, TCP Rebuilding, which enables a Web cluster to be content-aware and significantly improves our early work [17]. TCP Rebuilding allows the Web switch to transfer an established connection with a client to a chosen request-handling server by rebuilding the TCP connection in the request-handling server. After the TCP connection has been rebuilt, the chosen request-handling server responds to the request from the client directly, bypassing the Web switch. In particular, TCP Rebuilding could rebuild the TCP connection at the request-handling server in accordance with the established connection once a request packet has received. That is, TCP Rebuilding uses only the original HTTP request packet for connection rebuilding, and no extra packets for connection transfer are required. Compared with previous studies [2–5,8,9,11,14–16] this could reduce traffic in the internal network of Web clusters when handling TCP connection transfers, reducing the load of the system and speeding up the responses of Web clusters.

DOI: 10.1002/spe

Based on our proposed TCP Rebuilding mechanism, we have designed and implemented an efficient content-aware Web cluster named LVS-CAD, i.e. a Linux virtual server (LVS) with content-aware dispatching (CAD). We make use of the source codes of the LVS cluster [18–20] because of its popularity, stability, and high performance. By integrating the TCP Rebuilding mechanism into the LVS cluster, LVS-CAD enables the Web cluster to perform content-aware request distribution. Furthermore, to efficiently support HTTP/1.1 persistent connections [21], the multiple TCP Rebuilding mechanism similar to multiple TCP connection handoff [3] is proposed and implemented. Moreover, we have designed and implemented three content-aware request distribution policies based on Web page types and utilized a disk cache. The experimental results show that the performance of the Web clusters in processing static and dynamic requests has been significantly improved, especially for mixed or heavy loading requests.

The following sections first briefly introduce the background technologies and related work, then present the proposed TCP Rebuilding mechanism and its extension, the multiple TCP Rebuilding mechanism. The design and implementation of our LVS-CAD platform based on the TCP Rebuilding mechanism is then presented, and the proposed content-aware request distribution policies are introduced. Finally, the experimental results are reported and our conclusions are presented.

2. BACKGROUND AND RELATED WORK

We begin this section by introducing the content-blind dispatching platform, LVS, and then describe the existing TCP connection transfer mechanisms and content-aware request distribution policies. Finally, other related work is discussed.

2.1. LVS

The LVS [18–20] was developed to establish high-performance, highly available, and highly scalable Web clusters based on Linux. Recently, LVS has been widely adopted by enterprises [22] and has become part of the Red Hat Linux [23] distribution. The architecture of a Web cluster consists of a front-end server for dispatching and routing requests from clients and multiple back-end servers for actually handling requests.

LVS supports three packet-forwarding mechanisms, namely network address translation (NAT), IP tunneling, and direct routing [18,24]. From these mechanisms the most efficient is direct routing, in which requests from clients are first transmitted to the front-end and are then forwarded to back-ends, whereas requested data are transmitted directly from back-ends to clients, bypassing the front-end. Figure 1 shows the packet forwarding flow of the LVS using a direct routing mechanism. When the front-end receives the first SYN packet from the client, the LVS will immediately call the dispatching/scheduling module and select the back-end responsible for the packet. As this SYN packet contains only protocol headers without the HTTP contents, the dispatching/scheduling module will not be able to perform the 'content-aware' request distribution. Shortly afterwards, when the front-end receives the packets consisting of HTTP requests, it will directly forward these packets to the responsible back-end instead of performing a new scheduling. This is why the LVS cannot perform content-aware request distribution.

2.2. TCP connection transfer mechanisms

If a Web cluster needs to perform the content-aware request distribution, the scheduling timing of selecting a responsible back-end should be delayed until the front-end has received the request packet

DOI: 10.1002/spe

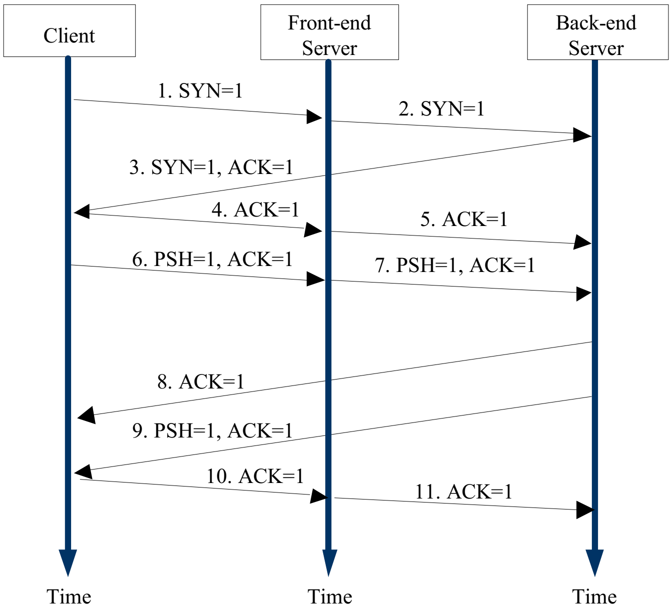


Figure 1. Packet forwarding flow of the LVS.

consisting of the HTTP contents. However, according to the TCP specification [25], the client will not transmit request packets until the connection has been established. For this reason, the front-end has to conduct three-way handshaking with the client. When the request packet has received and been processed, it will then be forwarded to the selected back-end.

In this section we introduce the existing TCP connection transfer mechanisms, including TCP splicing [15], redirect flows [16], TCP handoff [8], and one packet TCP state migration [14].

2.2.1. TCP splicing

As shown in Figure 2, the front-end establishes TCP connections with the client and the back-end. When receiving the packets from the client, the front-end will modify the source IP of the packets and changes the IP address to that of the front-end. Meanwhile, the front-end will change the destination IP of the packets into the IP address of the back-end. In addition, the front-end has to modify the sequence number and acknowledgment number in the TCP header, and recalculating checksum values of the IP and TCP layers in order to transfer the packets to the back-end.

When the TCP splicing mechanism applies to the Web clusters, two TCP connections must be maintained, which would result in an internal network full of redundant packets and inefficiency. In addition, making modifications to the inbound and outbound packets of the front-end would also affect the integral performance of the Web clusters.

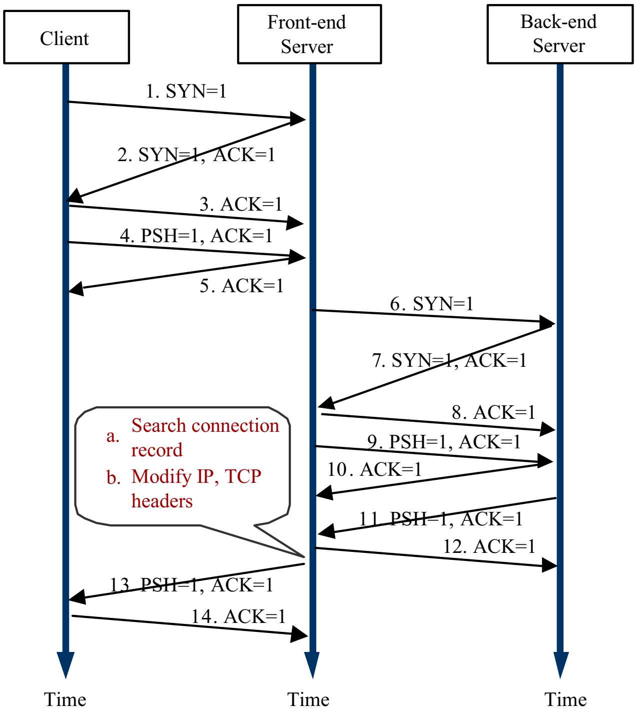


Figure 2. Packet forwarding flow of TCP splicing.

2.2.2. *Redirect flows*

Figure 3 shows that when the front-end receives the request packet from the client, it has to establish a new TCP connection with the back-end first, and then modify the acknowledgment number of the packet to correspond with the new TCP connection of the back-end. Finally, the front-end will forward the request packet to the back-end. The responses of the back-end will be forwarded to the front-end following the network architecture of NAT [24]. Then the front-end will modify the sequence number of the packets in accordance with the client's TCP connection first, and forward it to the client.

The redirect flows mechanism still has excessive resource consumption as a result of the extra packets transmitted for connection transfer and packet modification. Furthermore, the performance of the Web clusters is still limited by the NAT network architecture.

2.2.3. *TCP handoff*

As shown in Figure 4, the TCP handoff [8] mechanism utilizes its own custom protocol to transfer the TCP connection information between the client and the front-end to the back-end to establish a new TCP connection between the back-end and the client.

DOI: 10.1002/spe

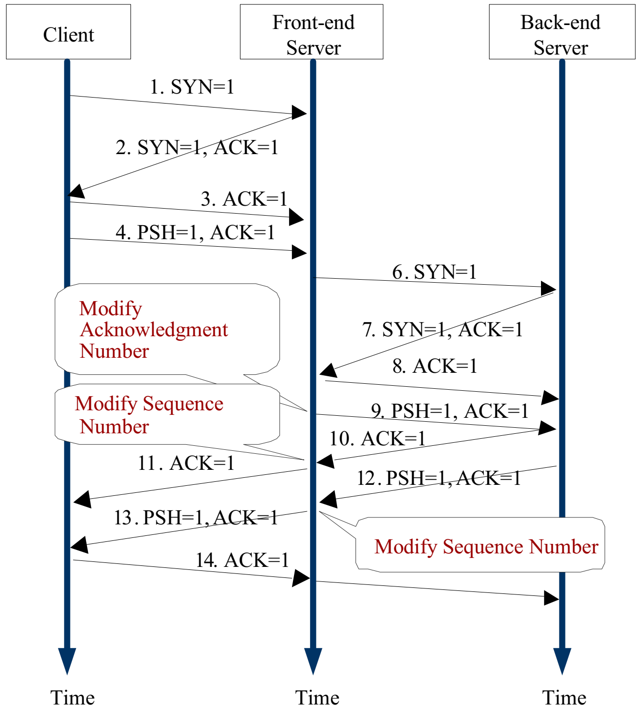


Figure 3. Packet forwarding flow of redirect flows.

While the back-end applies the TCP handoff mechanism, it could transmit the response packets to the client directly to avoid making packet modifications. However, if there are many requests from clients, the front-end has to transmit numerous extra packets to transfer TCP connections, which would result in an over-consumption of the internal network bandwidth.

2.2.4. One packet TCP state migration

The principle of the one packet TCP state migration [14] mechanism is that the request packet contains information regarding TCP connection, and the back-end will reconstruct a TCP connection with the client based on the information forwarded, rather than utilizing its own custom protocol.

As shown in Figure 5, when the back-end receives the request packet, the filter process implemented on the back-end will first send a mock SYN packet to the TCP module to execute the three-way handshaking and establish a new TCP connection, and then the request packet is forwarded to the TCP module for further processing. The filter process will also intercept the subsequent inbound or outbound packets of the back-end and modify the sequence number and acknowledgment number of these packets so that they correspond to the new TCP connection.

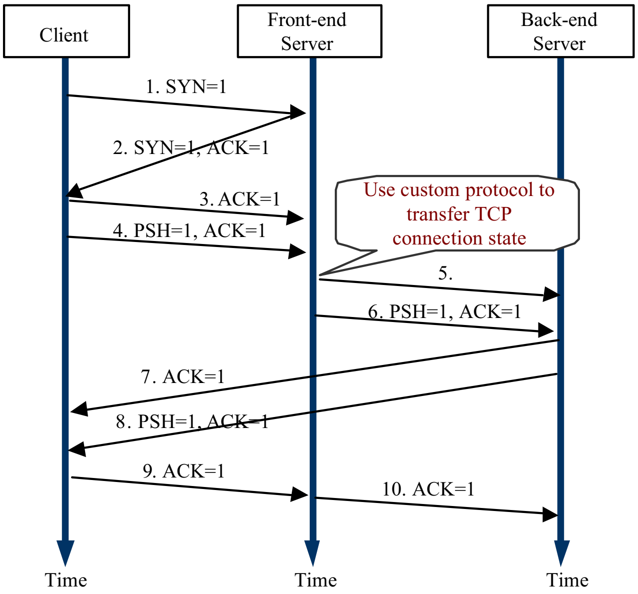


Figure 4. Packet forwarding flow of TCP handoff.

This mechanism allows the transfer of TCP connection by using a single received request packet. Consequently, it will efficiently reduce the number of transferred packets to prevent the congestion of the internal network. In addition to avoiding modifications to the kernel of the back-end, using the filter process to transfer the TCP connection will also efficiently solve the problem of session persistence. However, for the back-end, the one packet TCP state migration mechanism still requires mock packets for the three-way handshaking, and the filter process still has to proceed with the interception of and modifications to the inbound/outbound packets. These factors would affect the efficiency of the back-end.

2.3. Existent content-aware request distribution policies

In locality-aware request distribution with replication (LARD/R) [8], back-ends are grouped into server sets to service requested Web pages. Requested packets for the same Web page will be dispatched to the particular back-end set. If the target Web page is not served by any node of the server set, the least-loaded node of all of the back-ends will be selected to serve this packet. If the target Web page has been served by nodes of the server set, the least-loaded node of the server set will be assigned to serve this packet. Furthermore, the system will dynamically adjust the number of back-ends in the server set according to the loads of the back-ends. The concept of this approach is to increase the cache-hit ratio of back-ends and reduce disk I/O, and to improve response time as well as the performance of Web clusters. Therefore, LARD/R is suited to Web clusters with static requests or small memory [3,8].

DOI: 10.1002/spe

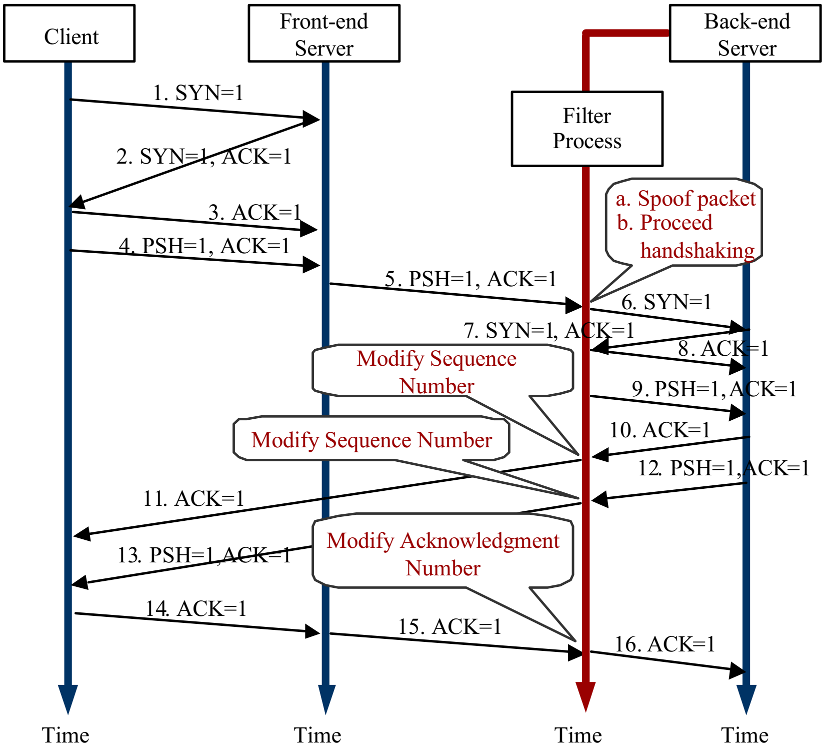


Figure 5. Packet forwarding flow of one packet TCP state migration.

The client-aware dispatching policy (CAP) [6] is another content-aware request distribution policy. The main goal of CAP is to improve load sharing in Web clusters that provides multiple types of services. The Web switch classifies requests from clients into four classes: normal, CPU bound, disk bound, and CPU and disk bound. When a HTTP request packet arrives, the Web switch first parses the URL and distinguishes which class it belongs to. It then schedules the packet in this class with the round-robin algorithm to produce an even distribution at the back-ends. However, requests with the same type might consume different amounts of resources, which would cause an inaccuracy in the back-ends' conjectured loading.

2.4. Other related work

Some studies [10,14] aim to support content-aware request distribution in the LVS. The TCPHA project [10], which is a subproject of LVS, also implements a one-way kernel-level layer-7 front-end for Linux and supports persistent connection so that every HTTP request is distributed taking into account its content and the response is directly sent to the client by the back-end. Furthermore, it implements TCP handoff inside the Linux kernel and stores files in back-ends by file type, so that it distributes requests by their types. The mechanisms of one packet TCP state migration and cookie name rewriting to packet filter [14] support content-aware request distribution, persistent connection, and session persistence in the LVS. The filter process is needed in each back-end to intercept and modify the inbound or outbound packets of the back-end, which would affect the efficiency of the back-end. In addition, it also belongs to one-way kernel-level layer-7 implementation.

The cluster architecture presented in [11] uses a two-way layer-7 front-end for dispatching requests in Linux. To transfer a TCP connection efficiently, the front-end pre-forks a number of TCP connections to each server and these pre-forked connections will be bounded to user connections for connection transfer. Based on this architecture, URL formalization was proposed to effectively support request distribution and reliability issues are discussed in a later work [12]. HACC [13] is also a two-way layer-7 cluster architecture designed for locality enhancement and dynamic load balancing. In a two-way cluster architecture, the inbound and outbound packets of back-ends should be modified and passed through the front-end, which would consume system resources of the Web cluster.

ClubWeb-1w [1,2] is a cluster architecture that uses the one-way layer-7 Web switch based on the TCP handoff approach. A new communication protocol, i.e. THOP, is implemented in the standard TCP/IP stack in Linux for TCP connection transfer. Performance results demonstrate that with careful design and implementation, a content-aware Web switch can be extremely scalable. However, additional packet transmission for connection transfer is still needed, which would consume cluster resources. Furthermore, handling for persistent connections was not discussed in that paper.

For handling HTTP/1.1 persistent connections by letting the front-end assign HTTP requests in the same connection to different back-ends, the back-end request forwarding mechanism [3] combines the TCP single handoff protocol with the forwarding of requests and responses among back-ends. In this mechanism, a connection can be handed off to a back-end only once. If the subsequent requests from the same connection cannot or should not be served by the designated back-end, it then forwards the request to another back-end or requests the content or service in question directly from another back-end selected by the front-end, and forwards the response back to the client. However, this approach requires data to be forwarded to the client through the designated back-end. This would waste CPU, memory, and network bandwidth at the cluster. So the back-end request forwarding mechanism is appropriate for requests that result in relatively small amounts of response data.

For HTTP/1.1 connection with the back-end request forwarding mechanism, extLARD [3] enhances the LARD [8] policy to consider forwarding overhead and disk utilization of the connection handling back-end in determining whether the request should be forwarded. Our work also enhances the LARD. However, for HTTP/1.1 connection with the multiple TCP connection handoff mechanism, we add the consideration of handoff overhead and disk I/O overhead in the selection of a back-end to serve the request.

For the scalability of content-aware request distribution in Web clusters, in the cluster architecture presented in [4], the TCP connection establishment and handoff are distributed over all back-ends, rather than being centralized in the front-end. Their architecture uses a layer-4 front-end for dispatching incoming requests to back-ends, where the request dispatching policies do not consider the requested content. The chosen back-end may forward the incoming request to another back-end by handing off the connection using the TCP handoff protocol to another back-end based on the requested content. Based on this cluster design, the workload-aware request distribution (WARD) strategy [7] assigns a small set of the most frequently accessed files to be served locally by any back-end, while partitioning the rest of files to be served by different back-ends. This was intended to reduce the forwarding or handoff overhead. Cyclone [9] is a similar cluster architecture that uses a layer-4 front-end for dispatching clients' requests. It makes use of socket cloning to allow an opened socket to be moved efficiently between back-ends and hot object replication for replicating frequently accessed documents to other back-ends. Therefore, the back-end chosen by the front-end will use socket cloning to clone the socket to the back-end that has the cache copy. For persistent HTTP connections in the above two works, subsequent requests of the same connection should be routed to the original connection handling back-end, which may add processing overhead and system resource usage at the cluster.

Half-pipe anchoring [26] is a technique that enables efficient multiple connection handoff. It decouples a TCP connection between a client and a cluster into two half-pipes, and anchors the half-pipe from the client to the cluster (control pipe) at a designated server while allowing the half-pipe from the cluster to the client (data pipe) to migrate on a per-request basis to an optimal server best suited to service the request. A light-weight communication protocol called Split-stack is devised to coordinate the control pipe and the data pipe. Handoff messages must be exchanged between servers in the cluster if handoff is required. The implementation and experiments on Linux demonstrate that this technique is efficient and scalable.

3. PROPOSED TCP CONNECTION TRANSFER MECHANISM: TCP REBUILDING

In this section we first present the TCP Rebuilding mechanism, and then, in order to efficiently handle requests in HTTP/1.1 persistent connections, multiple TCP Rebuilding is proposed and presented.

3.1. TCP Rebuilding

The TCP Rebuilding mechanism proposed in this research is an innovative mechanism for TCP connection transfer. Figure 6 shows the packet-forwarding flow in TCP Rebuilding, which is similar to that of the one packet TCP state migration mechanism as shown in Figure 5. However, there is a significant difference in the design of the back-ends. The TCP connection rebuilt by the TCP Rebuilding mechanism in the back-end agrees completely with the connection established previously by the client. During the TCP connection transfer, the TCP Rebuilding mechanism only needs to deal with the handshaking. When the connection has been rebuilt, the back-end can process the subsequent packets with a standard TCP operation procedure, so extra processing such as packet rewriting or modification for these subsequent packets will not be required.

Therefore, modifications to the TCP module in the kernel of the back-end are required. The related functions of three-way handshaking are modified to allow a new TCP connection to be established with the client using the information from the request packet.

The way that connections are rebuilt by the TCP Rebuilding mechanism is as follows. When the back-end receives the request packet from the front-end, the modified functions for three-way handshaking will be directly called to handshake with its own TCP module and initialize a new TCP connection in accordance with the established TCP connection at the client. Then the back-end could accept this request packet and transmit a response directly to the client.

Figure 7 shows the connection transfer handling diagram for TCP Rebuilding. The left-hand side of the figure is the process sequence for the TCP module in which the front-end handshakes

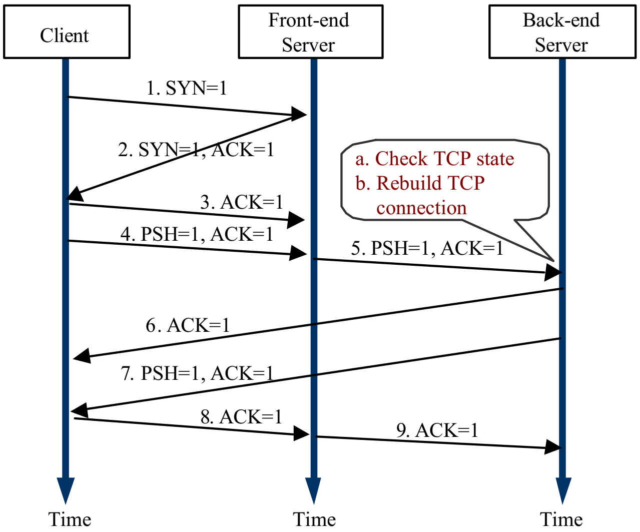


Figure 6. Packet forwarding flow of TCP Rebuilding.

with the client, while the right-hand side is the process sequence of the back-end's TCP module that rebuilds connection with TCP Rebuilding mechanism. The processing steps are detailed as follows.

1. The front-end receives the SYN packet from the client, and starts to proceed with the three-way handshaking.
2. The front-end calls SYN packet-handling functions in the TCP module.
3. The front-end initializes the sequence number, generates the SYN-ACK packet, and then transmits to the client.
4. The front-end receives the ACK packet from the client.
5. The front-end calls the ACK packet-handling functions in the TCP module. At this moment, the three-way handshaking is complete.
6. When the front-end receives the PSH packet from the client (the request packet that contains HTTP contents) it will call the request-scheduling module of the Web clusters to select a request-handling back-end, and then forward the packet to this selected back-end.
7. When the back-end receives the forwarded PSH packet, it starts to rebuild the TCP connection with TCP Rebuilding mechanism.
8. To spoof the TCP module at the back-end for receiving the SYN packet (as in Figure 1, step 2) that is physically non-existent, the back-end has to make a conjecture for the sequence number of the mock SYN packet from the sequence number of the PSH packet, and then set the MSS value to 1460.
9. The back-end calls modified handling functions for SYN packet to spoof the TCP module for receiving SYN packet.

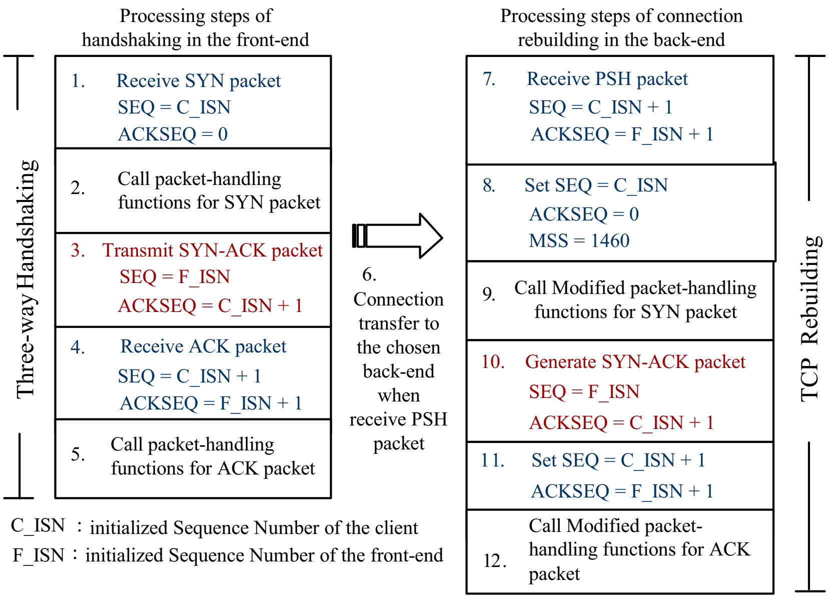

Processing steps of handshaking in the front-end
Processing steps of connection rebuilding in the back-end
Three-way Handshaking
1. Receive SYN packet SEQ = C_ISN ACKSEQ = 0
2. Call packet-handling functions for SYN packet
3. Transmit SYN-ACK packet SEQ = F_ISN ACKSEQ = C_ISN + 1
4. Receive ACK packet SEQ = C_ISN + 1 ACKSEQ = F_ISN + 1
5. Call packet-handling functions for ACK packet
6. Connection transfer to the chosen back-end when receive PSH packet
7. Receive PSH packet SEQ = C_ISN + 1 ACKSEQ = F_ISN + 1
8. Set SEQ = C_ISN ACKSEQ = 0 MSS = 1460
9. Call Modified packet-handling functions for SYN packet
10. Generate SYN-ACK packet SEQ = F_ISN ACKSEQ = C_ISN + 1
11. Set SEQ = C_ISN + 1 ACKSEQ = F_ISN + 1
12. Call Modified packet-handling functions for ACK packet
TCP Rebuilding
C_ISN : initialized Sequence Number of the client
F_ISN : initialized Sequence Number of the front-end

Figure 7. Connection transfer handling diagram for TCP Rebuilding.

10. To spoof the TCP module for transmitting the SYN–ACK packet (as in Figure 1, step 3) that is physically non-existent, the back-end has to make a conjecture for the sequence number of the mock SYN–ACK packet from the acknowledgment number of the PSH packet, and then calls modified handling functions to generate the SYN–ACK packet. However, this SYN–ACK packet will not be transmitted to the client to avoid resetting the client's connection.
11. To spoof the TCP module for receiving an ACK packet (as in Figure 1, step 5), the back-end has to make a conjecture for the sequence number and acknowledgment number of the mock ACK packet from sequence number and acknowledgment number of the PSH packet.
12. The back-end calls the ACK packet-handling functions to spoof the TCP module for receiving an ACK packet. At this point, TCP connection has been successfully rebuilt.

An example of the TCP connection transfer with the TCP Rebuilding mechanism is shown in Figure 8. Because the TCP Rebuilding mechanism only needs a request packet, which is originally indispensable to the back-end for responding, rather than packet rewriting or filter process to rebuild the connection at the back-end, connection transfer is thus efficient. Furthermore, subsequent packets can be handled by standard TCP operation procedures without the extra overhead due to packet modification.

Although this mechanism would result in a loss of window size and MSS information in a SYN packet, the dynamic window size can be acquired from subsequent packets and the MSS value can

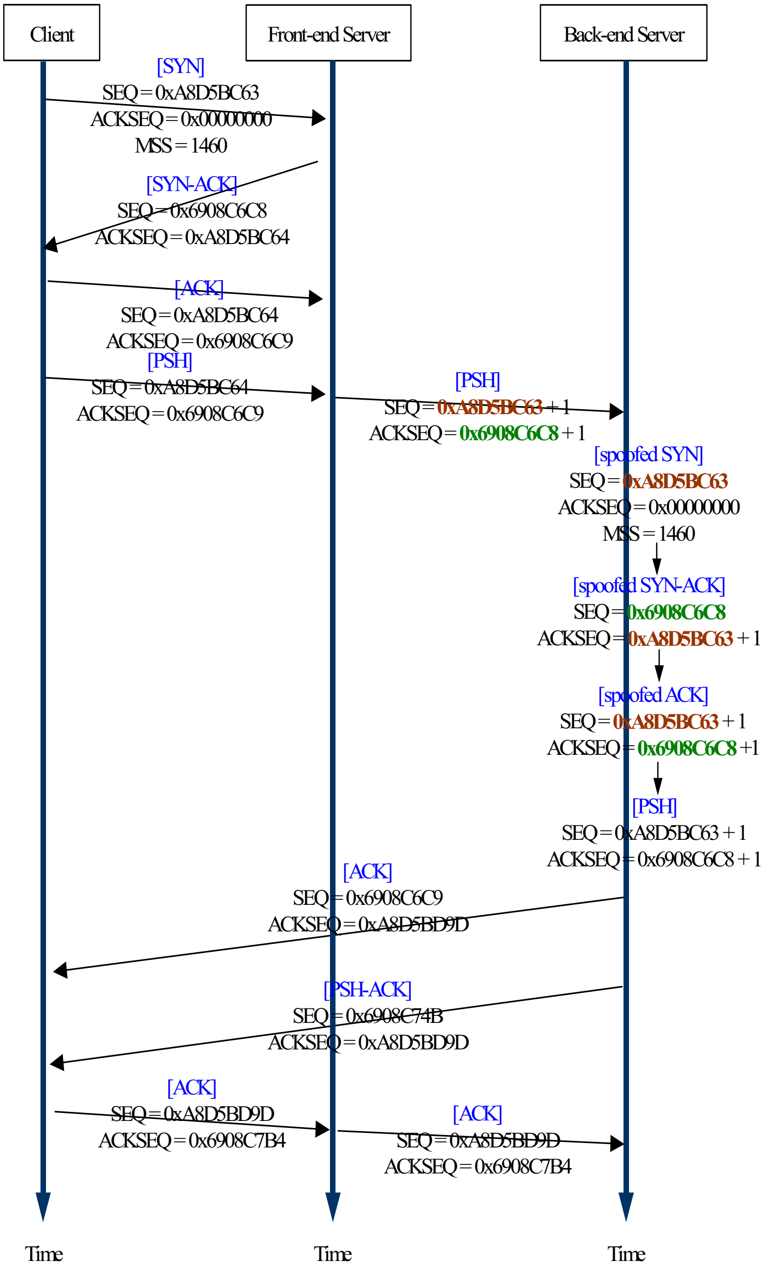


Figure 8. Example of the TCP connection transfer with the TCP Rebuilding mechanism.

DOI: 10.1002/spe

also be set to the standard value of 1460. An alternative approach is to put this information in the IP option of the original request packet sent to the back-end. Likewise, no extra packets are required for connection transfer.

Both the client and the back-end will jointly maintain the reliability of TCP connection. If the request packet drops in the forwarding process from the front-end to the back-end, the client will retransmit it while the TCP module is in timeout. In this way, together with the handshaking between the client and the front-end, the reliability of the whole connection process can be guaranteed. Furthermore, the TCP Rebuilding mechanism can also completely support the overall function of HTTP/1.1.

However, concerning session persistence, the proposed TCP Rebuilding should also provide a session affinity support in such a way that all requests that match a particular criteria can be directed to the same server. As session persistence is usually implemented by cookies and our front-end is content-aware, the front-end can redirect requests based on cookies to solve this problem. Because TCP Rebuilding belongs to a one-way implementation in which the response packets of back-ends are directly sent back to clients, bypassing the front-end, cookie name rewriting [14], which rewrites cookie names at back-ends to carry switching information, is very efficient and can be applied to support session persistence in TCP Rebuilding.

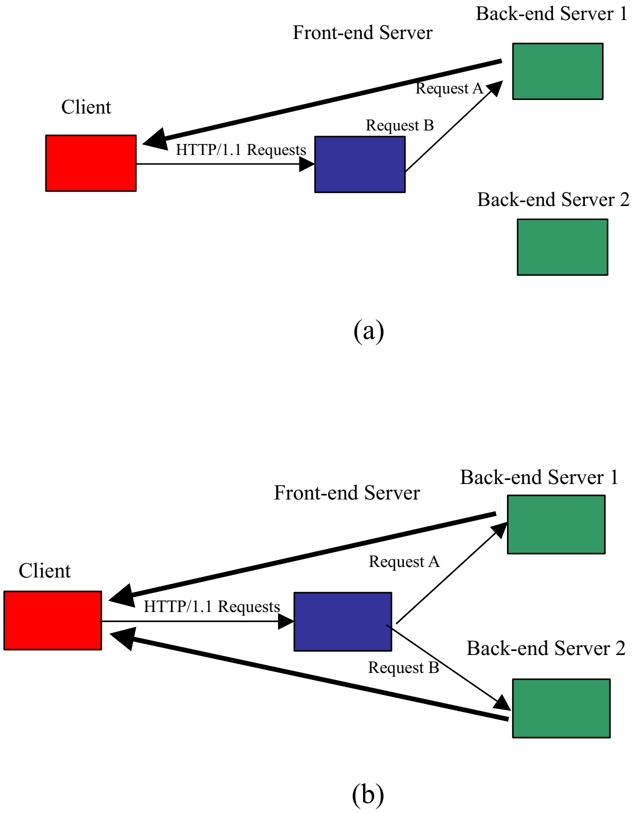
3.2. Multiple TCP connection rebuilding

For Web systems, if HTTP/1.1 persistent connection is applied to transfer packets, several service requests could be contained in a single TCP connection, which would reduce the cost of building a connection for each request and thus improve the performance of Web systems. However, for Web clusters, if multiple subsequent requests are all transmitted through the same connection to the same back-end, loads among back-ends would be unbalanced.

As shown in Figure 9(a), because the connection handling mechanism in LVS-CAD is designed based on LVS, the front-end will call the request scheduling module to generate a new connection record when it receives the first request packet. The subsequent request packets from the same client in the same TCP connection will be forwarded to the same back-end in the connection record instead of rescheduling. This will result in load unbalancing while the LVS-CAD adopts HTTP/1.1. Therefore, we propose the multiple TCP Rebuilding mechanism, which is similar to the multiple TCP connection handoff mechanism [3], in order to improve the connection handling in LVS-CAD for persistent connection.

In the multiple TCP Rebuilding mechanism, request scheduling is performed per request rather than per connection. As shown in Figure 9(b), the front-end receives two request packets from the client in an HTTP/1.1 connection. The scheduling module first analyzes request packet 1, then forwards it to the selected back-end 1 and rebuilds the connection using the TCP Rebuilding mechanism. The scheduling module will also schedule request packet 2. If the packet is scheduled to the back-end 2, the front-end will terminate the TCP connection at back-end 1, and the connection will be rebuilt at back-end 2. If the packet is still scheduled to back-end 1, no extra procedure is required. This enables the scheduling module to analyze the content of all request packets and request scheduling is performed per request instead of per connection, which will efficiently improve the accuracy of the estimation among back-end loading.

DOI: 10.1002/spe



(a)

(b)

Figure 9. Request distribution in HTTP/1.1 connection: (a) the HTTP/1.1 connection; (b) multiple TCP Rebuilding.

4. THE PROPOSED CONTENT-AWARE WEB CLUSTER: LVS-CAD

This research applies the TCP Rebuilding mechanism to the LVS [18–20] to produce a CAD platform called LVS-CAD, i.e. LVS with CAD, and to allow design dispatch scheduling based on the request types. To incorporate the TCP Rebuilding mechanism, not only is the IPVS module at the front-end replaced by the new IPVS-CAD module, but the TCP module at the back-end is also modified.

The architecture of LVS-CAD is the same as that of the LVS, utilizing the front-end to distribute service requests to the back-ends. As it shares the same architecture, LVS-CAD is also able to support the existing request scheduling algorithms and packet-forwarding mechanisms, including NAT, IP tunneling, and direct routing [18,24], built into the LVS.

In the following sections the three major parts of the LVS-CAD platform will be introduced: the implementation of fast TCP module handshaking that deals with the handshaking between TCP

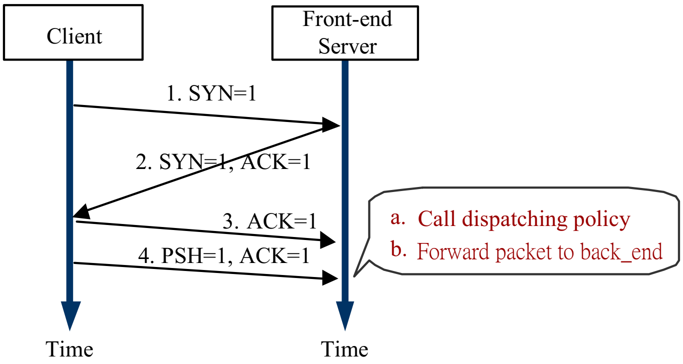


Figure 10. Packet flow of LVS-CAD with TCP module handshaking.

modules at the client and the front-end; TCP Rebuilding that deals with TCP connection transfer from the front-end to the back-end; and multiple TCP Rebuilding that provides efficient support of persistent connection.

4.1. Fast TCP module handshaking

For the front-end to perform content-aware request distribution, it must first perform three-way handshaking with the client to establish a TCP connection with the client before acquiring the contents of request packets from clients. To provide an efficient response, the front-end implements fast TCP module handshaking, which completes three-way handshaking in the IP layer instead of in the TCP layer. As shown in Figure 10, the procedures of TCP module handshaking in LVS-CAD are implemented efficiently, as described by the following packet sequence.

1. The client transmits a SYN packet to the front-end to request that a TCP connection be established, and then proceeds with three-way handshaking.
2. When the front-end receives the SYN packet, the IPVS module will immediately make a judgment. If the destination port of this packet is serviced by LVS-CAD, the IPVS module will generate a SYN-ACK packet based on a SYN packet and then transmit it to the client.
3. The client responds to the SYN-ACK packet with an ACK packet. At this moment, the TCP status of the client has changed to ESTABLISHED. However, the front-end will not deal with this ACK packet but drop it directly.
4. The client transmits the request packet consisting of HTTP contents to the front-end. The front-end then calls the request scheduling module to analyze the contents of this packet and select one back-end to process this request.

The implementation of this fast TCP module handshaking is summarized as follows: a SYN-ACK packet is directly produced by the IPVS-CAD module from the information contained in the SYN packet, and it is then transmitted to the client. The responsive ACK packet from the client

DOI: 10.1002/spe

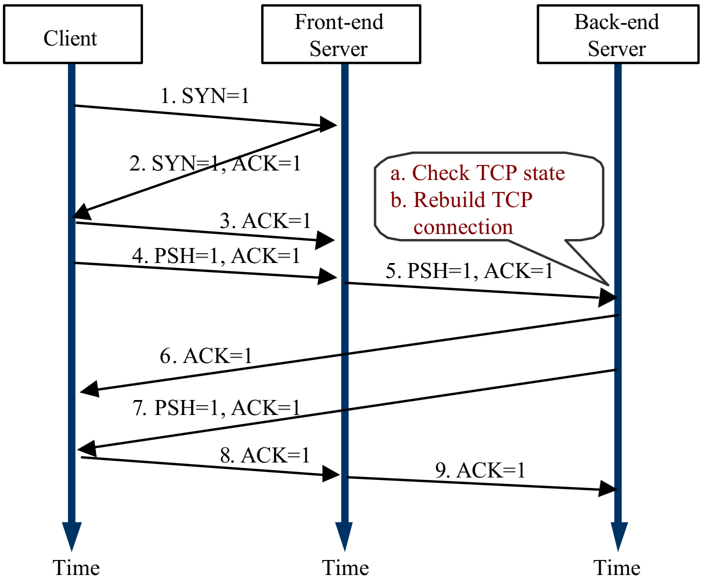


Figure 11. Packet forwarding flow of LVS-CAD with TCP Rebuilding.

is directly dropped. The merit of this method is that the whole process of handshaking at the front-end will be completed in the IP layer instead of in the TCP layer, so that the response time can be reduced.

4.2. TCP connection transfer

After the TCP module handshaking, the front-end must transfer the TCP connection to the back-end to complete the service request. Figure 11 shows how the TCP Rebuilding mechanism is applied to the TCP module at the back-end.

When the back-end receives the request packet forwarded from the front-end, the back-end will call the modified TCP functions and utilize the information in the request packet to spoof its own TCP module to rebuild the TCP connection with the client before the request packet is handled. After that, subsequent packets within the connection can be handled by the standard TCP operation procedure.

4.3. Implementation of multiple TCP Rebuilding

The implementation of the multiple TCP Rebuilding mechanism in the IPVS module includes three main parts: the first is used to correct connection information between the front-end and back-end; the second is used to maintain the loading information among back-ends by the new variable, total_load; and the third is used to terminate the unnecessary TCP connection at the back-end when that connection has been transferred. The loading value of the request will be added to the variable total_load of the selected back-end, and removed when this request is done.

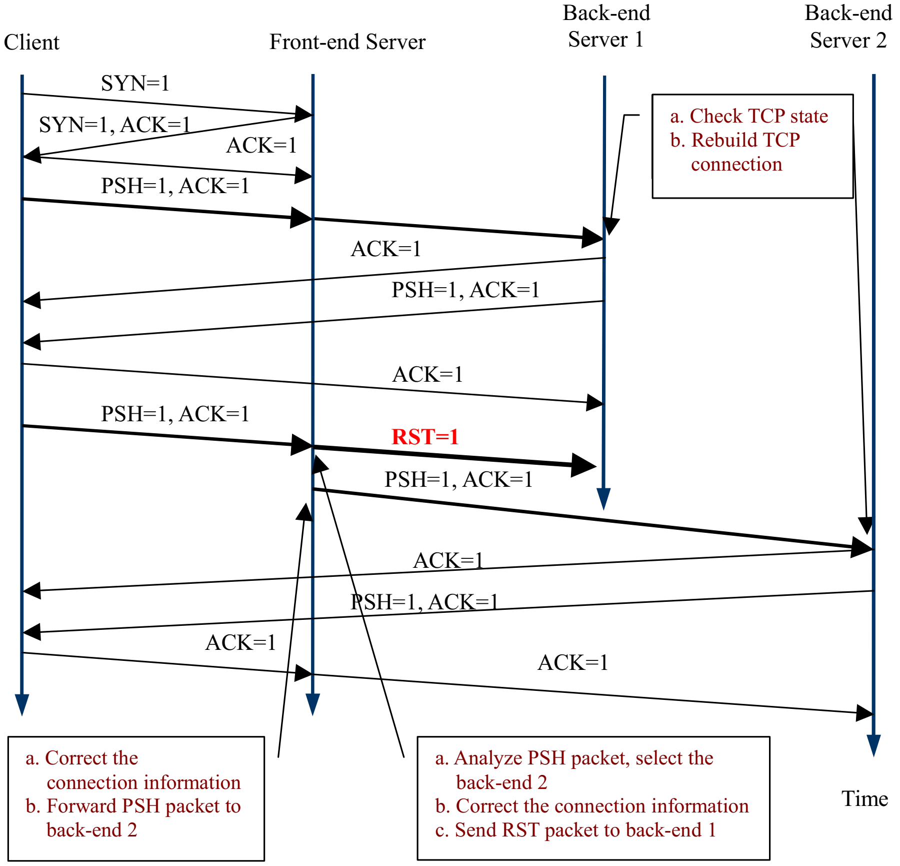

Client
Front-end Server
Back-end Server 1
Back-end Server 2
SYN=1
SYN=1, ACK=1
ACK=1
PSH=1, ACK=1
a. Check TCP state
b. Rebuild TCP connection
ACK=1
PSH=1, ACK=1
ACK=1
PSH=1, ACK=1
RST=1
PSH=1, ACK=1
ACK=1
PSH=1, ACK=1
ACK=1
ACK=1
a. Correct the connection information
b. Forward PSH packet to back-end 2
a. Analyze PSH packet, select the back-end 2
b. Correct the connection information
c. Send RST packet to back-end 1
Time

Figure 12. Packet forwarding flow for LVS-CAD with multiple TCP Rebuilding.

For the termination of the TCP connection, as shown in Figure 12, we use an RST packet to terminate the TCP connection that is not required at the back-end. When the front-end receives request packet 2, the RST packet will be created based on the header information of this packet to correspond with the TCP connection at back-end 1. After transferring the RST packet, request packet 2 will be forwarded to back-end 2.

5. PROPOSED CONTENT-AWARE REQUEST DISTRIBUTION POLICIES

This section presents the proposed content-aware request distribution policies, including content-aware weighted least load (CAWLL), extended locality-aware request distribution with replication policy (xLARD/R), and content-aware hybrid request distribution (CAHRD).

5.1. CAWLL policy

Scheduling algorithms built into the LVS will identify the loading of back-ends only by the number of connections. However, the loading has a strong bias against the number of connections, especially when the requests are mixed with static and dynamic Web pages and multiple requests may be transmitted through the same TCP connection for a HTTP/1.1 persistent connection. Therefore, based on the content of the request, we devise a scheduling algorithm called content-aware weighted least load scheduling algorithm (CAWLL) to select the back-end with weighted least load.

Each type of request has a different loading value, which represents the resource consumption. For simplicity, these values are set in our experiments according to the measured average number of response packets for handling these types of requests. The weight will be set for each back-end when the back-ends are heterogeneous with different hardware equipment.

CAWLL will be based on the content of the PSH packet to add the loading values and the RST packet to remove the loading values to maintain the loading information of all back-ends.

5.2. xLARD/R policy

The basic LARD/R strategy [8] considers only the current loading and current assignment of Web pages to back-ends for the selection of a back-end to service the given request. We then propose xLARD/R, which is modified from LARD/R to further consider the extra cost during the connection handoff.

As all requests in HTTP/1.1 persistent connection might be forwarded to the same back-end, which would cause unbalanced loading among the back-ends, cache management and multiple TCP connection handoff or multiple TCP Rebuilding mechanisms are thus required for LARD/R and xLARD/R.

However, the cost of the connection handoff is critical to the performance while multiple TCP Rebuilding or multiple TCP handoff mechanism is applied to a content-aware dispatching Web cluster, for example, LVS-CAD. Handoff TCP connections among the back-ends frequently result in vast resources being consumed and performance degradation. Therefore, three types of costs during the connection will be defined to determine the loading of back-end servers.

- *Request cost*. This is caused by the request itself. In our experiments, the request types and their corresponding costs are defined according to the measured average number of response packets for handling these types of requests.
- *Handoff cost*. This is the cost of the connection rebuilding during the handoff. It is set to 0 if the same back-end is selected to serve the given request. It is determined by experiment. In our experiments, it is set to be 20.
- *Disk I/O cost*. This is the extra cost when the back-end does not cache the requested Web page into the memory. It is set to 0 if the requested Web page is cached. It is determined by experiment. In our experiments, it is defined to be ten times the request cost.

As shown in Figure 13, we also define three formulas to determine the loading of back-end servers. The $load_A$ formula will be used if the requested Web page is cached in the server set responsible for serving this request. The $load_B$ formula will be used if the requested Web page is not cached in the server set responsible for serving this request. The $load_C$ formula will be used to find the back-end server with heaviest loading in the server set responsible for serving this request.

DOI: 10.1002/spe

$\mathit{total\_load} = \textit{current loading of the back-end}$
$\mathit{load}_A = \mathit{total\_load} + \textit{(Request Cost)} + \textit{Handoff Cost}$
$\mathit{load}_B = \mathit{total\_load} + \textit{(Request Cost)} + \textit{Handoff Cost} + \textit{Disk I/O Cost}$
$\mathit{load}_C = \mathit{total\_load} + \textit{(Request Cost)}$

Note: In many cases the Request Cost can be ignored to simplify the computation.

Figure 13. xLARD/R loading metrics.

```
1.   while true
2.     Fetch next r;
3.     if serverSet[r.request] = 0 then
4.        $n$.serverSet[r.target] $\leftarrow$ {node with least $load_B$};
5.     else
6.        $n \leftarrow$ {serverSet[r.target] node with least $load_A$ };
7.     if ($n$ is null) ||
8.        ($n$. $load_A > n$.weight && there is a node $l$ with $l$. $load_C < l$.weight/2) then
9.           $p \leftarrow$ { node with least $load_B$ };
10.          serverSet[r.target] = $p$;
11.          $n \leftarrow p$;
12.    if | serverSet[r.target] | > 1 &&
13.             time()- serverSet[r.target].lastMod > $K$ then
14.          $m \leftarrow$ {serverSet[r.target] node with most $load_C$};
15.          remove $m$ from serverSet[r.target];
16.    send r to $n$;
17.    if serverSet[r.target] changed then
18.       serverSet[r.target].lastMod $\leftarrow$ time();
```

Figure 14. xLARD/R pseudo code.

The pseudo code of xLARD/R is shown in Figure 14 and the detailed procedures are described below.

1–2: The system fetches the next request packet.

3–4: If the target Web page is not served/cached by any node of the server set, the least-loaded node of all back-ends, which is determined by $load_B$, will be selected to serve and added to this server set. The $load_B$ formula could be replaced by $load_A$ to simplify the computing.

5–7: If the target Web page is served/cached by nodes of the server set, the least-loaded node of the server set, as determined by $load_A$, will be assigned to n.

8–11: If n is overloaded, i.e. the $load_A$ of n is higher than its assigned weight, and there is a node with light load, i.e. the $load_C$ of the node is lower than half of its assigned weight, the least-loaded node of all back-ends, which is determined by $load_B$, will be assigned to p, which will then be added to this server set and assigned to n.

12–15: If the number of the nodes inside the server set is greater than one and the current time minus the last modified time of the server set is greater than K, then m, the most-loaded node of the

DOI: 10.1002/spe

```
if dynamic request
    Scheduling by CAWLL method;          // connection handoff cost is considered
else
    Scheduling by xLARD/R method;
```

Figure 15. CAHRD pseudo code.

server set, as determined by $load_C$, will be removed from the server set. K is defined in the same way as in LARD/R.

16–18: The system sends the requested packet to the node n and sets the last modified time of the server set to the current time.

5.3. CAHRD

If we consider the content of most Web sites we see that there are usually fewer dynamic Web pages than static Web pages, but that these dynamic Web pages cause heavier loading in the back-end. When xLARD/R is applied, dynamic requests might be dispatched to the same node, which would result in heavy loading in the back-end.

Therefore, we propose a hybrid method CAHRD that combines the advantages of CAWLL and xLARD/R. As shown in the pseudo code shown in Figure 15, dynamic requests have been separated while scheduling. CAWLL will be applied to dynamic requests and xLARD/R for others. However, to avoid unnecessary connection handoff in CAWLL, the connection handoff cost is also considered in the selection of the back-end with lightest load.

6. EXPERIMENTAL RESULTS

This section presents the experimental results for the LVS-CAD platform with the proposed CAWLL, xLARD/R, and CAHRD dispatching policies. To demonstrate the increase in efficiency achieved by incorporating the TCP Rebuilding mechanism into the LVS we compare the performance of LVS-CAD with the LVS.

6.1. Experimental environment

Table I shows our experimental hardware and network environment. In order to compare the cluster performance, we used the same configuration for both the LVS and LVS-CAD in all experiments. Each cluster consists of nine PCs, one for the front-end and eight for back-ends. A further six PCs are used as the clients for sending requests, and one of these serves as the controller. All of the experiments were performed in a private network to avoid being obstructed by the external network. All PCs are connected to a DLink DES-3225G switch and the packet forwarding mechanism of LVS-CAD and LVS is set to direct routing [18,24].

Table I. Hardware and network environment.

	CPU	Memory (MB)	HD (RPM)	NIC
Front-end	P4 2.4 GHz	DDR 256	7200	Reltek RTL8139
Back-ends 1–8	P4 2.4 GHz	DDR 256	7200	Reltek RTL8139
Controller/Client 1	P4 1.7 GHz	DDR 256	7200	SiS900 PCI F-E
Client 2	P3 800 MHz	SDRAM 256	7200	SMC
Client 3	P3 1.0 GHz	SDRAM 256	7200	Accton EN1207F
Client 4	P3 2.0 GHz	DDR 256	7200	SMC
Client 5	P3 2.4 GHz	DDR 256	7200	Reltek RTL8139
Client 6	P3 2.4 GHz	DDR 256	7200	D-link

Table II. Software environment.

	OS	Kernel	IPVS	Web server	Benchmarks
Front-end	Red Hat Linux 8.0	2.4.18	1.0.4	Apache 2.0.40	—
Back-end	Red Hat Linux 8.0	2.4.18	—	Apache 2.0.40	—
Controller/Client	Windows XP	SP1	—	—	WebBench 5.0
Client	Windows XP	SP1	—	—	WebBench 5.0

Table II shows the experimental software environment. All of the PCs in the LVS-CAD cluster have Red Hat Linux installed, the front-end uses IPVS for request dispatching, and the back-ends have Apache [28] for HTTP service installed. HTTP/1.1 connection is applied. In addition, all clients have Windows XP and WebBench [27] installed for request proposing.

WebBench is a performance testing software for Web servers, including both the controller and clients. The controller is able to control clients for proposing requests, to record and summarize the experimental data, and then output the experimental results. In addition, WebBench can control the mixed ratio of request types transmitted from clients by the programmable workload.

We performed all experiments to analyze the system under the heaviest workload that the system under test can sustain. As we want to evaluate system performance under different ratios of request types (e.g. different localities of hot Web pages) we also create a workload generator to generate a synthetic workload for various ratios of request types. The requested Web pages that are derived and enlarged from WebBench's default workload are 340 MB in total, and 13 kB on average. The performance metrics we used are the requests per second (req s^{-1}) and megabits per second (Mbps), which are the experimental results summarized and reported by WebBench.

6.2. Platform performance

This evaluation compares the performance of a CAD platform, i.e. LVS-CAD, with that of a content-blind dispatching platform, i.e. the LVS. This experiment is used to determine whether the performance

DOI: 10.1002/spe

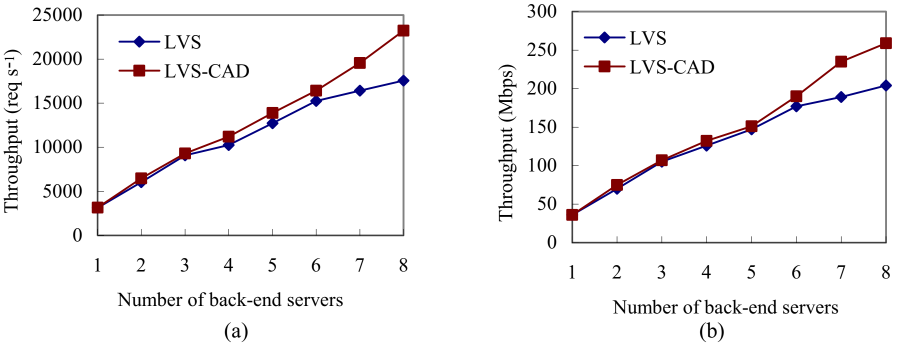


Figure 16. Platform performance for the LVS versus LVS-CAD: (a) throughput in requests per second; (b) throughput in megabits per second.

of a CAD platform can be comparable to a content-blind dispatching platform. Both platforms use the same content-blind weighted least connection (WLC) request scheduling algorithm. In addition, WebBench is used and the assigned workload is limited to one requested Web page that is 1 kB and can be fully cached in the memory of each back-end to avoid the effect of disk I/O. The number of back-ends ranges from one to eight.

Figure 16 shows that LVS-CAD performs 0.8–32.4% better than the LVS. These results show that even when the content-blind request scheduling algorithm is used, applying the TCP Rebuilding mechanism and fast TCP module handshaking can improve the performance of the LVS, and that this setup is more scalable when the number of back-ends has increased. Furthermore, dispatching requests according to their contents can be achieved in LVS-CAD.

6.3. Analysis of scalability

This experiment compares the scalability of the LVS cluster with content-blind dispatching and the LVS-CAD cluster with CAD. In this experiment the WLC request scheduling algorithm is applied to the LVS for comparison, and the proposed content-aware request distribution policies, i.e. CAWLL, LARD/R, and xLARD/R, are applied to LVS-CAD. WebBench is used and the enlarged workload, i.e. 340 MB, is adopted for all clients, with the number of back-ends ranging from one to eight.

As shown in Figure 17, LVS-CAD with xLARD/R performs 58.7–156.49% better than LVS/WLC when the number of back-ends is larger than one. However, LVS-CAD with CAWLL gives a lower level of performance than LVS/WLC. There are two reasons for the lower performance of CAWLL: first, the resource consumption during the connection handoff is not considered by CAWLL; second, the frequent connection handoff causes cache misses among the back-ends.

6.4. Different localities of hot Web pages

Hot Web pages are Web pages with the higher levels of pageview. In this test, WebBench is used and hot Web pages are built from the requested Web pages of the default workload.

DOI: 10.1002/spe

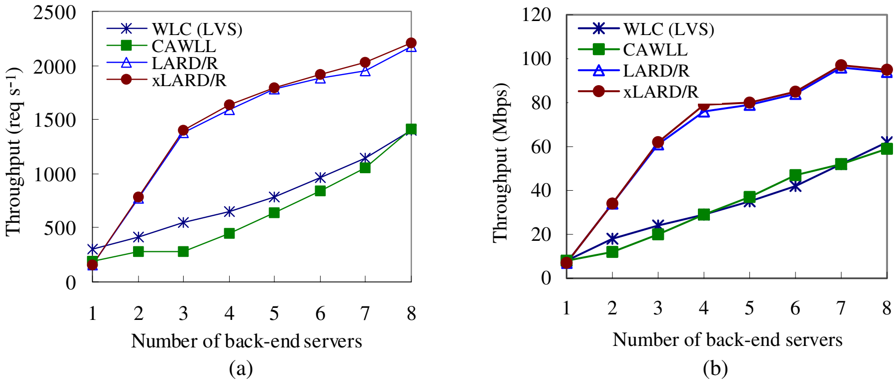


Figure 17. Scalability Test for the LVS versus LVS-CAD: (a) HTTP/1.1 throughput in requests per second; (b) HTTP/1.1 throughput in megabits per second.

Furthermore, we prepare the click through rate (CTR) with 20%, 40%, 60%, and 80%, and change the percentage of hot Web pages in requested Web pages to 10%, 20%, 30%, and 40%.

WLC is applied to the LVS, while CAWLL, LARD/R, and xLARD/R are applied to LVS-CAD to examine the performance of each scheduling algorithm/platform pair. Eight back-ends are used in this test. As shown in Figure 18, TCP connection transfer and cache miss would still cause a substantial performance degradation to CAWLL in LVS-CAD. Compared with LVS/WLC, LVS-CAD with xLARD/R gives a performance speedup of 14.37–64.28%, and LVS-CAD with LARD/R gives a performance speedup of 8.01–59.01%.

In conclusion, LVS-CAD with xLARD/R gives the best performance under all CRTs. This result demonstrates that reducing the cost of handoff and the disk I/O can efficiently improve the performance of Web clusters.

6.5. Static versus dynamic content

In this test, WebBench is used. The enlarged workload used in Sections 6.3 and 6.4 consists of complete static Web pages. To examine the impact of dynamic Web pages on performance, the static Web pages have been mixed with dynamic Web pages written with PHP in the ratio of 90%:10% (denoted as 90s10d), 80%:20% (denoted as 80s20d), 70%:30% (denoted as 70s30d), and 60%:40% (denoted as 60s40d). Eight back-ends are used in this test.

As shown in Figure 19, LVS-CAD with CAHRD gives the best performance. Compared with LVS/WLC, LVS-CAD with CAHRD gives a performance speedup of 13.26–36.72%, LVS-CAD with xLARD/R gives a performance speedup of 2.83–24.16%, and LVS-CAD with LARD/R gives a performance speedup of 0.94–17.24%.

The 70%:30% ratio is the critical point for xLARD/R and CAWLL. Before this point xLARD/R gives a better performance than CAWLL, and after this point CAWLL performs better than xLARD/R. This also indicates that CAWLL is more suitable for a workload with dynamic Web pages.

DOI: 10.1002/spe

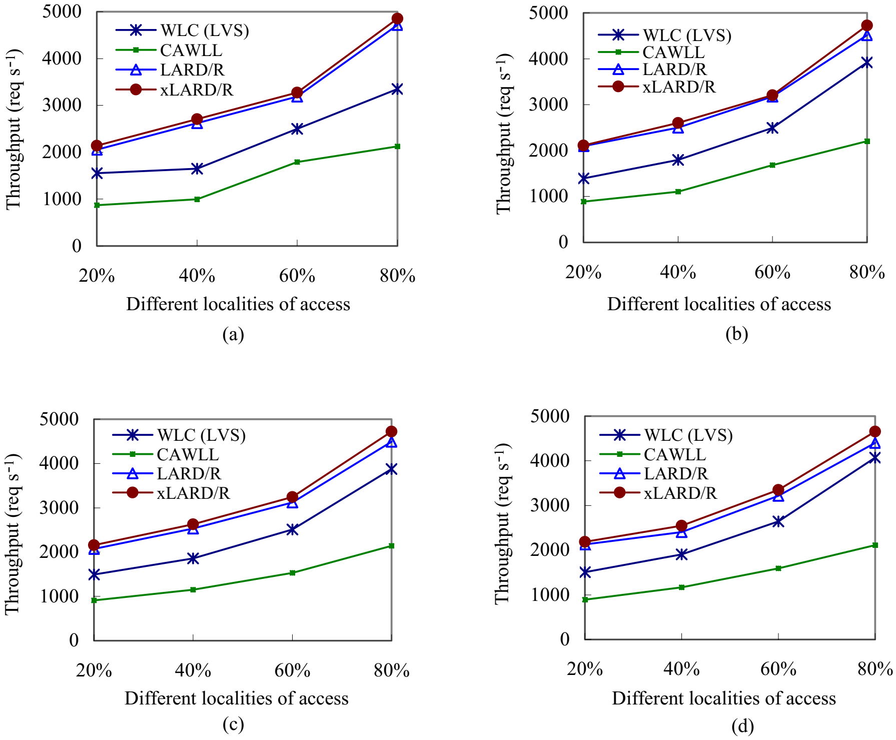


Figure 18. Performance of different localities of hot Web pages for the LVS versus LVS-CAD: (a) 10%; (b) 20%; (c) 30%; (d) 40%.

By combining the advantages of xLARD/R and CAWLL, CAHRD gives the best performance of these request scheduling algorithms.

7. CONCLUSIONS

We have designed and implemented a CAD platform, LVS-CAD, to build efficient and scalable Web clusters. LVS-CAD can extract and analyze the content in requests, then dispatch each request by its content or type of service requested. By applying the proposed connection transfer mechanism, TCP Rebuilding, the connection between the dispatching server and the request-handling server can be transferred efficiently. By applying the proposed content-aware request distribution policies to reduce the frequent connection handoff and increase the cache hit ratio of Web servers, request distribution among Web servers is more balanced and the system resources of the overall Web cluster are thus utilized more efficiently. Together with the implementation of the fast TCP module handshaking and

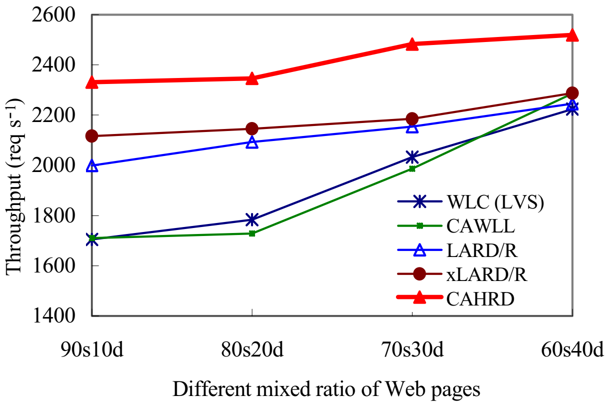


Figure 19. Performance of mixed ratio of dynamic and static contents for the LVS versus LVS-CAD.

multiple TCP Rebuilding, LVS-CAD provides efficient support for HTTP/1.1 persistent connection in Web clusters with content-based request distribution.

Performance results indicate that LVS-CAD outperforms the content-blind dispatching platform, the LVS. By applying TCP Rebuilding and fast TCP module handshaking only, LVS-CAD can improve on the performance of the LVS by 32.4% and is more scalable as the number of back-ends is increased. Together with the employment of the proposed content-aware request distribution policies, LVS-CAD with xLARD/R performs 58.7–156.49% better than LVS/WLC for scalability test and 14.37–64.28% better for different localities of hot Web pages. LVS-CAD with CAHRD outperforms LVS/WLC by 13.26–36.72% for a mixed ratio of dynamic and static Web pages.

Based on this platform, several issues can be explored or enhanced in future research, such as the cooperation of cache management among back-end servers, session persistence handling, providing differentiated service, quality of service support, special Web content deployment, efficient methods for analyzing the content of requests, etc. In addition, pre-establishing TCP connections between the front-end and back-ends would be helpful for reducing the TCP connection transfer time.

REFERENCES

1. Andreolini M, Colajanni M, Nuccio M. Kernel-based Web switches providing content-aware routing. *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, 16–18 April 2003. IEEE Computer Society Press: Los Alamitos, CA, 2003; 25–32.
2. Andreolini M, Colajanni M, Nuccio M. Scalability of content-aware server switches for cluster-based Web information systems. *Proceedings of the 12th International World Wide Web Conference*, Budapest, Hungary, 20–24 May 2003. The International World Wide Web Conference Committee (IW3C2), 2003.
3. Aron M, Druschel P, Zwaenepoel W. Efficient support for P-HTTP in cluster-based Web servers. *Proceedings of the Annual Usenix Technical Conference*, Monterey, CA, June 1999. USENIX Association: Berkeley, CA, 1999.
4. Aron M, Sanders D, Druschel P, Zwaenepoel W. Scalable content-aware request distribution in cluster-based network servers. *Proceedings of the Annual Usenix Technical Conference*, San Diego, CA, 18–23 June 2000. USENIX Association: Berkeley, CA, 2000.

5. Cardellini V, Casalicchio E, Colajanni M, Yu PS. The state of the art in locally distributed Web-server systems. *ACM Computing Surveys* 2002; **34**(2):263–311.
6. Casalicchio E, Colajanni M. A client-aware dispatching algorithm for Web clusters providing multiple services. *Proceedings of the 10th International World Wide Web Conference*, Hong Kong, 1–5 May 2001. The International World Wide Web Conference Committee (IW3C2), 2001; 535–544.
7. Cherkasova L, Karlsson M. Scalable Web server cluster design with workload-aware request distribution strategy WARD. *Proceedings of the 3rd International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, San Jose, CA, 21–22 June 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 212–221.
8. Pail VS, Aront M, Bangat G, Svendsent M, Druschelt P, Zwaenepoelt W, Nahumq E. Locality-aware request distribution in cluster-based network servers. *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998. ACM Press: New York, 1998; 205–216.
9. Sit Y-F, Wang C-L, Lau F. Cyclone: A high-performance cluster-based Web server with socket cloning. *Cluster Computing* 2004; **7**(1):21–37.
10. Wang L. TCPHA Project Website, 28 April 2004. http://dragon.linux-vs.org/∼dragonfly/htm/tcpha.htm.
11. Yang C-S, Luo M-Y. Efficient support for content-based routing in Web server clusters. *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, 11–14 October 1999. USENIX Association: Berkeley, CA, 1999.
12. Yang C-S, Luo M-Y. System support for scalable, reliable, and highly manageable Web hosting service. *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, 26–28 March 2001. USENIX Association: Berkeley, CA, 2001.
13. Zhang X, Barrientos M, Chen JB, Seltzer M. HACC: An architecture for cluster-based Web servers. *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, 12–13 July 1999. USENIX Association: Berkeley, CA, 1999.
14. Lin Y-D, Tsai P-T, Lin P-C, Tien C-M. Direct Web switch routing with state migration, TCP masquerade, and cookie name rewriting. *Proceedings of the Global Telecommunications Conference 2003 (GLOBECOM '03)*, San Francisco, CA, 1–5 December 2003, vol. 7. IEEE Computer Society Press: Los Alamitos, CA, 2003; 3663–3667.
15. Maliz D, Bhagwat P. TCP splicing for application layer proxy performance. *Technical Report RC-21139*, IBM, March 1998.
16. Steven C, Krawczyk JJ, Nair RK, Royce K, Siegel KP, Stevens RC, Wasson S. Method and system for directing a flow between a client and a server. *US Patent 6,006,264*, 21 December 1999.
17. Liu H-H, Chiang M-L. TCP Rebuilding for content-aware request dispatching in Web clusters. *Journal of Internet Technology* 2005; **6**(2):231–240.
18. Linux Virtual Server Website. http://www.linuxvirtualserver.org/ [18 October 2006].
19. Zhang W, Jin S, Wu Q. Creating Linux virtual servers. *Proceedings of the 1999 LinuxExpo Conference*, Raleigh, NC, 18–23 May 1999. Available at: http://www.linuxvirtualserver.org/Documents.html.
20. Zhang W. Linux virtual servers for scalable network services. *Proceedings of the OTTAWA Linux Symposium*, Ottawa, Canada, 19–22 July 2000.
21. Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, Berners-Lee T. Hypertext Transfer Protocol—HTTP/1.1. *Request for Comments 2616*, 9 July 2004. Available at:
http://www.w3.org/Protocols/rfc2616/rfc2616.html [1 September 2004].
22. LVS Deployment, 9 July 2004. http://www.linuxvirtualserver.org/deployment.html [2 August 2006].
23. Red Hat. http://www.redhat.com/ [3 October 2006].
24. Mack J. LVS-HOWTO, July 2003.
http://www.austintek.com/LVS/LVS-HOWTO/ [25 February 2006].
25. Postel J. Transmission Control Protocol, *Request for Comments 793*, September 1981.
26. Kokku R, Rajamony R, Alvisi L, Vin H. Half-pipe anchoring: An efficient technique for multiple connection handoff. *Proceedings of the 10th International Conference on Network Protocols (ICNP 2002)*, Paris, France, November 2002; 12–21.
27. WebBench Web site. http://www.etestinglabs.com/benchmarks/webbench/webbench.asp [8 February 2005].
28. Apache. http://www.apache.org/ [3 October 2006].

DOI: 10.1002/spe