Direct Web Switch Routing with State Migration, TCP Masquerade, and Cookie Name Rewriting

Ying-Dar Lin, Ping-Tsai Tsai, Po-Ching Lin, and Ching-Ming Tien

Department of Computer and Information Science,
National Chiao Tung University, Hsinchu, Taiwan
E-mail: {ydlin, motse, pclin, juvenia}@cis.nctu.edu.tw

***Abstract*— Existing layer 4 load balancers are content-blind and often have difficulty in redirecting HTTP requests to the appropriate server in the session manner. Layer 7 load balancers, also referred to as web switches, are content-aware and support session persistence. However, most web switches employ a bi-direction architecture, which means both request and response traffic must both pass through the load balancer. This makes a web switch become a bottleneck easily. In this paper, we present a direct routing architecture to prevent response traffic from passing through the web switch. Our solution is highly scalable in the number of back-end servers. In addition, two simple but effective mechanisms, one-packet TCP state migration and cookie name rewriting to packet filter, are presented to support persistent connection and session persistence. Through the external benchmark, we prove that our system outperforms existing solutions. The internal benchmark investigates the bottlenecks of our system and suggests the future improvement.**

I. Introduction

The booming growth of World Wide Web has driven extremely high demand for powerful web servers that are able to handle a large number of user requests. Web clustering is a common solution to increasing the capacity of web services. A load balancer is located in front of a cluster of web servers and redirects requests to one of them according to the preconfigured policy. The policy allows the load balancer to choose the best server based on appropriate criteria, such as server load.

A load balancer can operate at layer 4 or layer 7, depending on whether it can make redirection decisions according to the request content. An L4 load balancer establishes two connections between the client and the server – one from the client to itself, and the other from itself to the server. It then keeps forwarding subsequent requests in the same connection to the same server. Such a solution is efficient because it only processes information at layer 3 and layer 4, which has a fixed length and is located in the fixed position of a packet. Nowadays, many web sites use server side scripts to generate dynamic content such as database queries and shopping carts. These scripts establish an application layer session across multiple TCP connections to keep user data on the web server for a period of time. However, the L4 load balancer is content-blind and may redirect these TCP connections to different servers, which results in errors because the user information is stored on the server that the user first accesses. A web switch, also referred to as an L7 load balancer, avoids this problem by redirecting requests according to the content at layer 7, such as URLs, cookies and SSL identifiers.

In some existing L7 load balancing solutions, such as TCP splicing [1] and flow redirection [2], the response traffic must pass through the load balancer and the capacity of the load balancer can become the bottleneck easily. TCP handoff [3] modifies the TCP/IP kernel stack on a server to support a proprietary protocol, which allows TCP state to be migrated from the load balancer to the server, but it is hard to implement and support persistent connections [4]. The goal of our work is to build a web switch that can bypass response traffic without the modifying TCP/IP module on the server and support persistent connections elegantly.

In this paper, we propose three mechanisms to solve the above issues: one-packet state migration, switch-back masquerading, and cookie name rewriting. We implement these methods as a Linux kernel module, and benchmark its external and internal performance.

II. Architecture Design

The application layer information, such as URLs, is not available until a connection has been established between the client and the web switch. To respond directly to the client, a server must have the TCP state migrated from the web switch, which requires OS kernel modification and a proprietary protocol. A different approach, called application layer proxy, avoids TCP state migration by establishing two connections on the web switch, one to the client and the other to the server. An application proxy acts as a bridge between the client and the server. This approach may not be able to handle a huge number of requests [5] because the response traffic, which has much more bytes than the request does, must pass through and be dealt with by the proxy, making the proxy a bottleneck easily. TCP splicing belongs to this category. A better solution is flow redirection, but the response traffic still has to pass through the web switch.

To design a highly scalable web switch, we reduce the load of the web switch by preventing response traffic from passing through it. One feasible solution to date is TCP handoff, which allows the TCP state to be migrated from the web switch to the server so that the client and the server can communicate directly without passing their traffic through the web switch. Figure 1 illustrates the process of TCP handoff. In step 5,

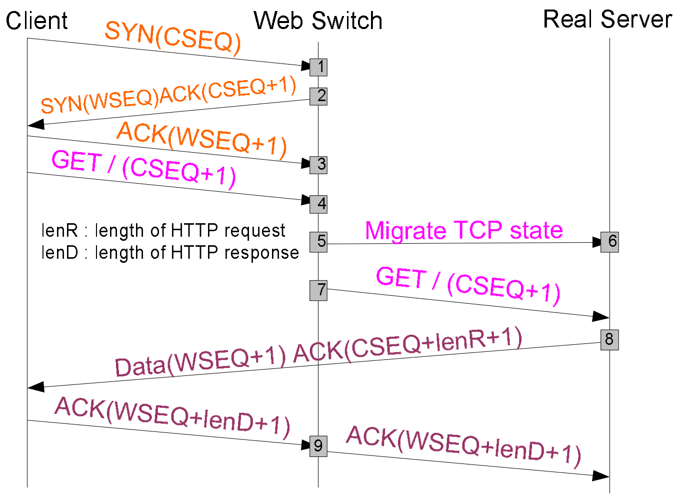


Fig. 1. Mechanism of TCP handoff.

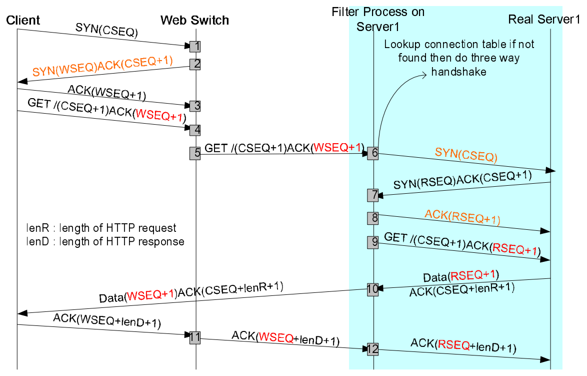


Fig. 2. Process of one-packet TCP state migration.

the established TCP state on the web switch is migrated to a server with a proprietary protocol. After the migration, the web switch sends the requests to the server. The server then can reply with data to the client directly since step 8.

TCP handoff is more scalable than TCP splicing. However, TCP handoff implementation requires modification in both OS kernel and applications [6], which is not always feasible. It is also hard to support persistent connections, and we explain it in Section III.

A. One-packet TCP state migration to packet filter

To avoid kernel modification, we propose a mechanism, one-packet TCP state migration to packet filter. We implement a packet filter that intercepts the connection from the web switch to the server without modifying the kernel. The mechanism is shown and discussed in Figure 2 and the following description.

- Steps 1-3: The client completes three-way handshake with the web switch.
- Steps 4-5: The client sends the request to the web switch. The web switch sends this request to the chosen server while keeping the ACK bit and the acknowledgment number WSEQ+1.

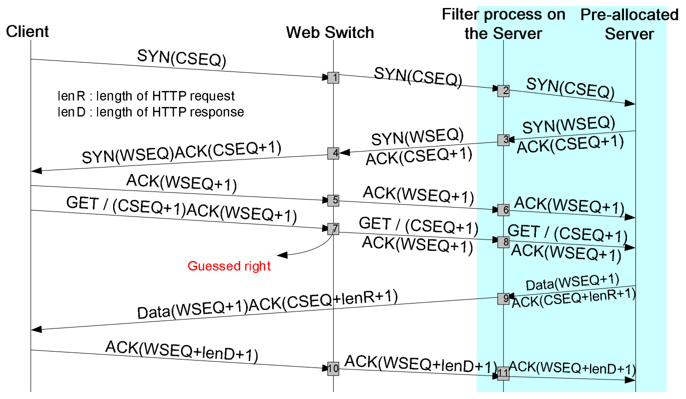
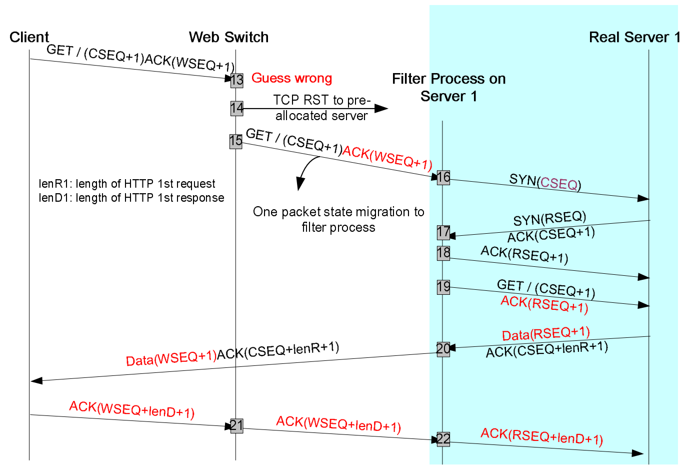

Client
Web Switch
Filter process on the Server
Pre-allocated Server
lenR : length of HTTP request
lenD : length of HTTP response
SYN(CSEQ)
SYN(CSEQ)
SYN(CSEQ)
SYN(WSEQ)ACK(CSEQ+1)
SYN(WSEQ) ACK(CSEQ+1)
SYN(WSEQ) ACK(CSEQ+1)
ACK(WSEQ+1)
ACK(WSEQ+1)
ACK(WSEQ+1)
GET / (CSEQ+1)ACK(WSEQ+1)
GET / (CSEQ+1) ACK(WSEQ+1)
GET / (CSEQ+1) ACK(WSEQ+1)
Guessed right
Data(WSEQ+1)ACK(CSEQ+lenR+1)
Data(WSEQ+1) ACK(CSEQ+lenR+1)
ACK(WSEQ+lenD+1)
ACK(WSEQ+lenD+1)
ACK(WSEQ+lenD+1)

Fig. 3. Pre-allocation scheme with a right guess.

- Step 6: The packet filter intercepts the request and starts to replay three-way handshake with the TCP module on the same server, using the same sequence number as that from the client.
- Step 7: The packet filter receives SYN+ACK with the sequence number RSEQ from the TCP module. The packet filter records RSEQ for TCP masquerading.
- Step 8-9: After completing three-way handshake, the packet filter changes the request's acknowledgment number to RSEQ+1 and sends it to the TCP module.
- Step 10: The packet filter changes the sequence number of the response packet from RSEQ+1 to WSEQ+1.
- Steps 11-12: The subsequent packets from the client to the web switch in the same connection will be redirected to the same server, and the packet filter will masquerade the sequence number in the packets between them. The client is totally oblivious of the transition that makes the connection with the web switch be inherited by the server.

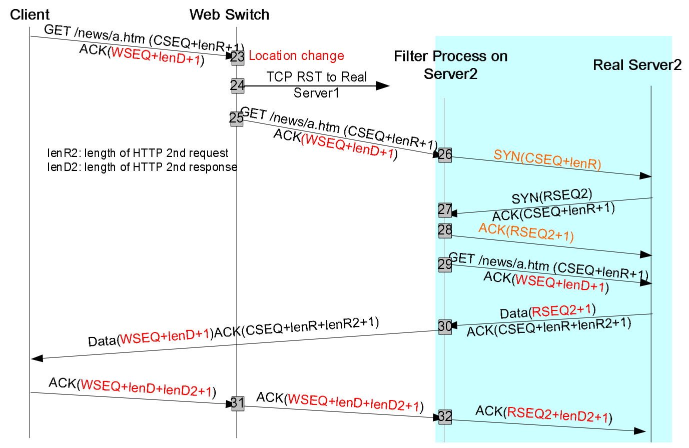
The above method is not highly scalable because the web switch has to play TCP three-way handshake for every new connection. For the scalability, we use a pre-allocation scheme, as shown in Figure 3 and Figure 4. When a client sends a SYN packet to the web switch, the web switch selects a server based on some heuristic first, say connection statistics. After receiving the HTTP request, the web switch knows if its selection is right (i.e., the requested content is on the selected server). If it is right, it passes the request to the pre-allocated server. If not, the web switch sends an RST packet to disconnect the pre-allocated connection, and then redirects the request to an appropriate server.

We notice a similar architecture by Chow [7], which also has a filtering concept and a pre-allocation scheme. However, our architecture supports persistent connections. Each request in a TCP connection must pass through the web switch. The web switch can redirect the request to another server if the content is located on that server, as described in Section III. If we delegate the whole TCP connection to the client and the server, the packet filter must be responsible for redirecting a request to the right server. Our architecture also allows reusing a previous connection because an RST packet instead of a FIN packet



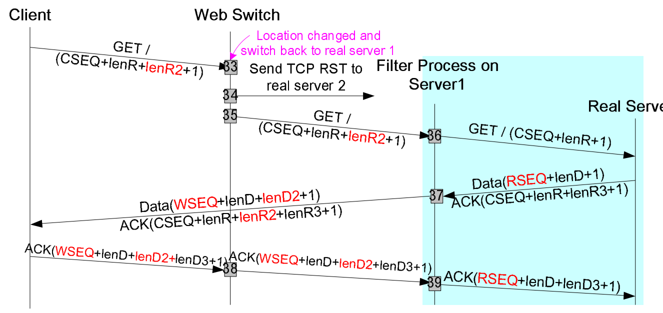
Client
Web Switch
Real Server 1
GET / (CSEQ+1)ACK(WSEQ+1)
Guess wrong
Filter Process on Server 1
TCP RST to pre-allocated server
GET / (CSEQ+1)ACK(WSEQ+1)
lenR1: length of HTTP 1st request
lenD1: length of HTTP 1st response
One packet state migration to filter process
SYN(CSEQ)
SYN(RSEQ) ACK(CSEQ+1)
ACK(RSEQ+1)
GET / (CSEQ+1) ACK(RSEQ+1)
Data(RSEQ+1) ACK(CSEQ+lenR+1)
Data(WSEQ+1)ACK(CSEQ+lenR+1)
ACK(WSEQ+lenD+1)
ACK(WSEQ+lenD+1)
ACK(RSEQ+lenD+1)
13
14
15
16
17
18
19
20
21
22

Fig. 4. Pre-allocation scheme with a wrong guess.



Client
Web Switch
GET /news/a.htm (CSEQ+lenR+1) ACK(WSEQ+lenD+1)
Location change
TCP RST to Real Server1
Filter Process on Server2
Real Server2
GET /news/a.htm (CSEQ+lenR+1) ACK(WSEQ+lenD+1)
lenR2: length of HTTP 2nd request
lenD2: length of HTTP 2nd response
SYN(CSEQ+lenR)
SYN(RSEQ2) ACK(CSEQ+lenR+1)
ACK(RSEQ2+1)
GET /news/a.htm (CSEQ+lenR+1) ACK(WSEQ+lenD+1)
Data(RSEQ2+1) ACK(CSEQ+lenR+lenR2+1)
Data(WSEQ+lenD+1)ACK(CSEQ+lenR+lenR2+1)
ACK(WSEQ+lenD+lenD2+1)
ACK(WSEQ+lenD+lenD2+1)
ACK(RSEQ2+lenD2+1)
23
24
25
26
27
28
29
30
31
32

Fig. 5. Process of the second request.



Client
Web Switch
Location changed and switch back to real server 1
GET / (CSEQ+lenR+lenR2+1)
Send TCP RST to real server 2
Filter Process on Server1
Real Server1
GET / (CSEQ+lenR+lenR2+1)
GET / (CSEQ+lenR+1)
Data(RSEQ+lenD+1) ACK(CSEQ+lenR+lenR3+1)
Data(WSEQ+lenD+lenD2+1) ACK(CSEQ+lenR+lenR2+lenR3+1)
ACK(WSEQ+lenD+lenD2+lenD3+1)
ACK(WSEQ+lenD+lenD2+lenD3+1)
ACK(RSEQ+lenD+lenD3+1)
33
34
35
36
37
38
39

Fig. 6. Process of the third request.

is sent to cease a connection (See Section III). Besides, it supports session persistence by cookie name rewriting, which is not described in Chow's architecture.

However, our solution still has two drawbacks in spite of its simplicity and high scalability. First, in a pipeline connection [8], which allows multiple requests to be sent before the response of a previous request returns, the acknowledgment number in the request may be that for another response. Fortunately, pipeline connections are rarely used and we can ignore this possibility when we refer to persistent connections herein. Second, TCP options such as SACK or ECN are negotiated during three-way handshake. All TCP options are decided in the connection between the client and the web switch, so all of the servers are required to support and have the same options as the web switch.

III. Supporting HTTP Persistent Connections

It is difficult for a web switch to support HTTP persistent connections because multiple requests in a connection may be assigned to different servers, but only the first assigned server has the TCP state from the web switch. The other servers do not know what to do upon receiving an unexpected HTTP request. Some existing solutions to the support are HTTP header rewriting, HTTP 302 redirection [8], and multiple TCP connection handoff and backend request forwarding [4].

A. Switch-back masquerading at packet filter

One-packet TCP state migration to packet filter uses the sequence number of a request to rebuild the TCP state, which can be used to solve the problem with persistent connections easily. For example, after the first request has been redirected to server 1, the second request in the same connection is sent to the web switch. The process is illustrated in Figure 5.

- Step 23: The web switch receives the request. After parsing the HTTP header, it notices the requested content is located on server 2.
- Step 24-25: The web switch sends RST to server 1 to close the connection and then redirects the request to server 2.
- Steps 26-28: When the packet filter on server 2 receives the request, it checks if the kernel has the TCP state of this connection. If not, the packet filter plays three-way handshake with the kernel to rebuild the TCP state. Otherwise, the packet filter masquerades the TCP sequence number and then sends the request to the kernel.
- Step 29: After rebuilding the TCP state, the packet filter masquerades the TCP sequence number and then sends the packet into the kernel.
- Step 30: The response packets pass through the packet filter and are de-masqueraded.
- Steps 31-32: The subsequent packets are redirected between the client and server 2 and are masqueraded by the packet filter.

If the content of the third request is again on server 1, the web switch can reuse the previous connection to it. To reuse this connection, we cannot close it in step 24. We send RST to tell server 1 to keep silent and take over the connection when the connection is switched back, as shown in Figure 6. When the packet filter receives RST, it stalls this RST for 15 seconds. During this period, if there are no more requests in this connection, the RST packet will be sent into kernel to close the connection. The web switch can send one more RST packet to close the connection to a server in the silent mode immediately. The process is illustrated in Figure 6.

- Step 33: The web switch receives the third request and

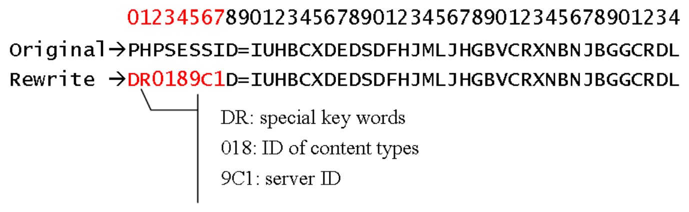


Fig. 7. Cookie name rewrite at packet filter.

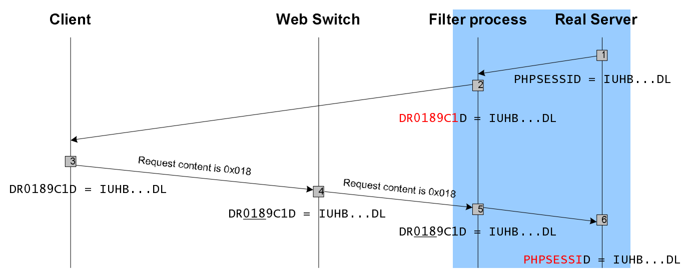


Fig. 8. Session persistence with multiple back-end servers - part 1.

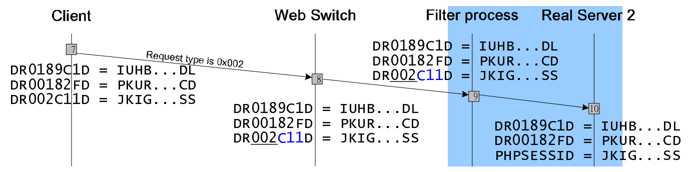


Fig. 9. Session persistence with multiple back-end servers - part 2.

decides to switch back to server 1.

- Step 34: The web switch sends an RST packet to tell server 2. Server 2 can close the connection later if no more requests are received.
- Step 35: The web switch sends the request to server 1.
- Step 36: Server 1 reuses the previous state without a new three-way handshake.
- Steps 37-39: The subsequent packets are masqueraded as usual.

IV. Supporting Session Persistence

Session persistence is usually implemented by cookies. When a server initiates a session, it inserts a unique session cookie. The requests from the same client in the following connections will carry the cookie and the server will use it to identify the client in different connections. Since a web switch is content aware, it can redirect requests based on cookies to solve this problem. Some existing solutions are pre-defined cookie name, automatic cookie insertion, and cookie learning.

A. Cookie name rewriting at packet filter

The forementioned solutions are not designed for direct routing. Automatic cookie insertion and cookie learning are infeasible because the response packets must pass through the web switch. Pre-defined cookie name can be used for direct routing, but it is not transparent to servers. All of them also have a common restriction that the session should persist in a single server. This disallows URL switching and session persistence to be coexistent. We hope to design a mechanism that allows the possibility. Our solution avoids modifying HTTP header length and packet size. When the response packets pass through the packet filter, the packet filter finds out the session cookie and rewrites the first eight characters of the cookie name to carry switching information. The format is illustrated in Figure 7. The first two characters are a keyword to indicate the cookie has been modified to carry switching information. The following three characters is the identifier of content types. The last three characters represent the server identifier. All following requests in the same session carry this special cookie, and the web switch redirects the request by the cookie name. The restriction is that the original cookie name should be at least eight characters long, which is acceptable in real situations. Figure 8 shows the details.

In step 2, the packet filter rewrites the first eight characters of the session cookie name, say ”DR0189C1”. This is unique to each server. In step 4, the web switch parses the URL for the requested content type, say ”0x018”. If it finds the cookie name with the first five characters ”DR018”, it retrieves the server identifier and redirects the packet to that server. Otherwise, it redirects based on the request content. In step 5, the packet filter searches the HTTP header to find the string ”DR0189C1” and writes the first eight characters of the session cookie name back to ”PHPSESSI”.

After several requests, the client may carry multiple cookies such as those shown in Figure 9. Our solution allows URL switching in cookie persistence.

V. Implementation and Benchmark

We integrate our mechanism into Linux Virtual Server (IPVS) [9], a popular open source project of L4 load balancer. IPVS is a Netfilter module in Linux kernel 2.4.x [10]. Our implementation consists of a DWSR web switch and DWSR packet filters on the servers. Figure 10 shows the functional blocks of the system. The shaded block on the left is the web switch, which has a packet filter module to intercept the request packets before they enter the TCP/IP module. Then, the dispatcher looks up the connection table and rule table to make a redirection decision. The MAC Changer module then changes the destination MAC address to that of the target server. The packet filter on the server intercepts the packet. If the server has the TCP state of the connection, the packet is passed to the TCP masquerading module to adjust the TCP sequence number. Otherwise, the filter replays three-way handshake with the kernel to rebuild the TCP state of the connection. The response packets pass through the TCP masquerading module to adjust the TCP sequence number, and then go to the client directly.

A. External Benchmark

We benchmark DWSR and compare it with KTCPVS [11] and some commercial products without the direct routing schemes. We use the same hardware platform, which is equipped with dual Athlon XP 1700+ CPUs and gigabit NICs, to execute both DWSR and KTCPVS. Six PCs equipped with

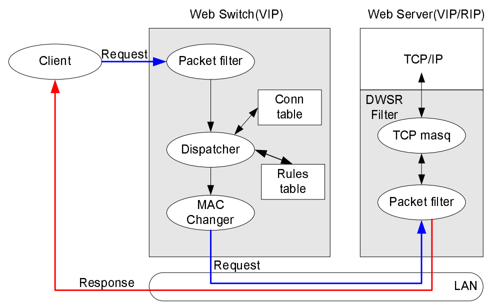


Fig. 10. Functional blocks of direct web switch routing system

TABLE I
RESULTS OF EXTERNAL BENCHMARK.

Throughput	Requests/second			Mbps		
Number of servers	2	4	6	2	4	6
DWSR with 100Mbps NIC	3,328	6,584	8,878	177	350	472
KTCPVS with 100Mbps NIC	1,865	1,866	1,866	89	89	89
KTCPVS with 1000Mbps NIC	3,210	5,639	5,654	170	298	295

Intel Pentium III 1 GHz CPUs act as the web servers, each of which runs Apache 1.3 and can service 1,700 requests/second in 85 Mbps of response traffic.

The result is shown in Table I. When KTCPVS, a non-direct routing architecture, comes with a 100 Mbps NIC, the six servers can inundate the NIC easily because one server alone can produce 85 to 89 Mbps of response. With a 1000 Mbps NIC, the performance of KTCPVS will boost, but the bottleneck becomes the CPU. In DWSR, the response bypasses the web switch, so neither the NIC nor the CPU is the bottleneck and the performance of the DWSR grows linearly. DWSR delegates most burdens, such as TCP masquerading and setting up connections with clients to the servers, so the performance is much better. Because of lack of enough clients and servers, we did not reach the upper-bound performance of DWSR. The time for DWSR to process a request is 69.723 μs, and we estimate an Intel Pentium III 1 GHz PC can process about 14,285 requests/second. We also benchmark some existing products in the same way, and the result is shown in Table II.

B. Internal Benchmark

To identify the bottleneck of DWSR, we measure how much time each function takes. The result is listed in Table III. For each request, DWSR may need to create a new connection entry, look up the entry if it has been created, or redirect

TABLE II
BENCHMARKING DWSR WITH EXISTING PRODUCTS.

Solution	Requests/second
DWSR	8,878
Cisco CSS 11154	6,020
KTCPVS	5,654
Nortel ACE Switch 180e	5,402
F5 Big-IP HA+Controller	5,151
Radware WSD	2,763
Foundry ServerIron XL	2,840

TABLE III
RESULTS OF INTERNAL BENCHMARK.

DWSR	Time	DWSR-filter	Time
IPVS	46 μs	Rebuild TCP state	101.39 μs
L7 overhead	24 μs	TCP masquerading	3.54 μs
Enabling cookie persistence	12 μs	Enabling cookie persistence	4.47 μs

it. This process totally takes about 46 μs on average. Layer 7 processing takes about 24 μs in which 43% of the time to generate a TCP RST packet, 40% to look up service entry and 17% to parse the file extension of the URL pattern. Content parser can be a bottleneck if we parse the entire content, which alone takes about 10 to 15 μs to search the URL pattern. Another bottleneck is to generate a TCP RST packet, but it only happens when the chosen server is changed.

The bottleneck of the DWSR filter is rebuilding the TCP state because it alone takes about 100 μs to replay three-way handshake with the kernel. Enabling cookie persistence takes much more time because it has to search the cookie and extract the destination server identifier from that cookie. A good pattern matching algorithm is required to speed up the process.

VI. CONCLUSION AND FUTURE WORK

We propose a direct routing architecture of web switch and show this architecture is indeed a scalable way to construct a web switch. One-packet TCP state migration to packet filter makes TCP state migrate to servers and support persistent connections easily. Cookie name rewriting is an easy and powerful idea to support session persistence when URL switching is enabled. However, some pending issues are not well solved, such as supporting SSL while URL switching is enabled and accelerating content rule matching. These topics deserve further study to build a fully-functional web switch.

REFERENCES

[1] D. Maltz, *TCP Splicing for Application Layer Proxy Performance*, IBM Research Report RC-21139, March 1998.
[2] Colby, and et al., *Method And System for Directing a Flow between A Client And A Server*, US patent 6,006,264, December 21, 1999.
[3] V. S. Pai, and et al., *Locality-Aware Request Distribution in Cluster-based Network Servers*, Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, Oct 1998.
[4] M. Aron, P. Druschel and W. Zwaenepoel, *Efficient Support for P-HTTP in Cluster-Based Web Servers*, USENIX 1999 Annual Technical Conference, Monterey, CA, June 1999.
[5] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, *Scalable Content-aware Request Distribution in Cluster-based Network Servers*, Proceedings of USENIX 2000 Annual Technical Conference, San Diego, CA, June 2000.
[6] *Scalable Content-aware Request Distribution in Cluster-based Network Servers*, http://www.crpc.rice.edu/softlib/scalableRD.html.
[7] C. Chow, *A tutorial presented in PDCAT 2001, Taipei Taiwan*, http://cs.uccs.edu/~chow/pub/conf/pdcat/tutorial.ppt.
[8] R. Fielding, and et al., *Hypertext Transfer Protocol - HTTP/1.1*, IETF RFC 2616, June1999.
[9] *Linux Virtual Server Project*, http://www.linuxvirtualserver.org/.
[10] *Netfilter*, http://www.netfilter.org/.
[11] *Kernel TCP Virtual Server (KTCPVS)*, http://www.linuxvirtualserver.org/software/ktcpvs/ktcpvs.html.