Fault-tolerant and scalable TCP splice and web server architecture

Manish Marwah **Shivakant Mishra**

Department of Computer Science,
University of Colorado,
Campus Box 0430, Boulder, CO 80309-0430.
Email: {marwah,mishras}@cs.colorado.edu

Christof Fetzer

Department of Computer Science,
Dresden University of Technology,
Dresden, Germany D-01062.
Email: cf2@inf.tu-dresden.de

CU Techinal Report CU–CS–1003–06

January 2006

Fault-tolerant and scalable TCP splice and web server architecture

Manish Marwah Shivakant Mishra
Department of Computer Science,
University of Colorado,
Campus Box 0430, Boulder, CO 80309-0430

Christof Fetzer
Department of Computer Science,
Dresden University of Technology
Dresden, Germany D-01062

Abstract

This paper proposes three enhancements to the TCP splicing mechanism: (1) Enable the same TCP connection to be simultaneously spliced through multiple machines for better scalability; (2) Make a spliced connection fault-tolerant to proxy failures; and (3) Provide flexibility of splitting the splicing functionality between a proxy and a backend server for further increasing the scalability of a web server system. A web server architecture based on this enhanced TCP splicing is proposed. This architecture provides a highly scalable, seamless service to the users with minimal disruption during server failures. In addition to the traditional web services in which users download webpages, multimedia files and other types of data from a web server, the proposed architecture supports newly emerging web services that are highly interactive, and involve relatively longer, stateful client server sessions. A prototype of this architecture has been implemented as a Linux 2.6 kernel module, and the paper presents important performance results.

1 Introduction

TCP splicing [20, 27] is popular for increasing the performance of serving web content through proxies. Organizations offering Internet services commonly use application layer proxies at the edge of their networks. These proxies perform a number of important functions, notably layer 7 or content-based routing, i.e. routing of client requests based on their content to an appropriate application server. Other functions performed by such a proxy include web content caching, implementation of security policies (e.g. authentication, access control lists), implementation of network management policies (e.g. traffic shaping) and usage accounting.

In an application level proxy, TCP splicing avoids copying of data between kernel and user space, and context switching, resulting in improved performance, and thus making the performance of a proxy comparable to IP forwarding [19]. References [26, 25, 11] describe the advantages of TCP splicing in web server architectures. At present, there are two drawbacks of a TCP splicing based web server architecture: (1) All traffic between clients and servers (both directions) must pass through the proxy, thus making the proxy scalability and performance bottlenecks; and (2) This architecture is not fault tolerant; if a proxy fails, clients have to re-establish their HTTP connections and re-issue failed requests (even if a backup proxy is present).

The work presented in this paper consists of two major parts. The first part consists of enhancements to TCP splice. We describe the design and implementation of a distributed and replicated architecture of TCP splice. This architecture allows a TCP connection to be spliced at multiple proxies, providing both fault-tolerance and higher scalability. In particular, the failure of a proxy causes no disruption to a splice; and additional proxy machines can be added to scale the system. For even higher scalability, the architecture allows for the splicing functionality to

be split between the proxies and a backend server, with the backend server performing the splicing functionality for the response packets.

An important consideration of our work is to realize a server architecture with inexpensive, off-the-shelf components. Although, use of specialized hardware, e.g. a network processor, can improve performance [32], its cost is an order of magnitude higher than a low-end, general-purpose processor[1], although the throughput increases only a few times.

The second component of this paper consists of using these TCP splice enhancements to build a flexible, scalable and fault-tolerant web server architecture. The nature of user services provided through the Internet is changing. User applications that usually run on a machine locally are beginning to be offered remotely through a web browser for ease of both accessibility from any location, and, of sharing information with others. The web browser is becoming more like a *remote desktop*, aided by techniques such as AJAX [1]. Some popular instances of such services are Google's Maps [14] and Gmail [13], Yahoo's Flickr [12], Microsoft's Virtual Earth [21]. Services similar to what MS Office suite provides (word processing, spreadsheet, etc.) are also beginning to emerge over a web browser, e.g. see Writely[31], Zohowriter[33], Writeboard[30], Num Sum[23], and so on.

There are two distinct features of these new types of applications that are not present in the traditional web services. The newer applications require relatively long-duration, stateful client server sessions. Furthermore, the amount of data flowing from the clients to the servers is also relatively large in such applications. As users become dependent on such services in increasing numbers and use them for critical tasks, failure of servers hosting such applications can cause significant disruption. For seamless user experience during server failures, it may not be sufficient to simply provide server fault-tolerance (e.g. through data replication and alternate servers). The *quality* of server fault-tolerance provided, in terms of the impact of the failure on the user, is also important. In some cases, it may not be acceptable if the recovery process takes tens of seconds. This, for instance, will be true if the application is real time, e.g. a stock market ticker, or, highly interactive in nature, e.g., a user browsing a roadmap on a web-based map service like Google Maps [14].

A large body of research exists on architecture of web server clusters which focuses mainly on scalability and efficiency [4, 5, 15, 17]. Fault-tolerance is usually assumed to come for free by virtue of using a cluster and replication of data. In the event of a server failure, the client is expected to retry its request. Although this model works well for conventional HTTP applications (displaying web pages to the client), it is not very effective for real-time, highly interactive web applications such as ones described above.

Based on the enhanced TCP splicing functionalities, we propose a web server architecture with a focus on enhanced server fault-tolerance while being as scalable and efficient as earlier such server architectures [5, 17]. It supports content-based request distribution (layer 7 routing), including support for HTTP 1.1 persistent connections. The enhanced fault tolerance results in minimal user impact in the event of server failures with outages not lasting more than a few seconds.

To summarize, the main contributions of this work are two fold: (1) Contributions in TCP splicing mechanisms; and (2) Enhancements in the design of a flexible web server architecture. It makes the following contributions to TCP splicing in general:

- Design and implementation of a replicated TCP splice.
- Design and implementation of a fault-tolerant TCP splice.
- Splitting a single TCP splice functionality into two unidirectional splices with segments in the two directions being spliced at different machines.

These enhancements are used as a basis for a web server architecture with the following properties:

- Highly scalable and fault-tolerant web server architecture.

[1] several thousand compared to a few hundred

- Fast, seamless and client transparent failovers with minimal impact on users during server failures.
- Well suited to providing server fault-tolerance for interactive and real-time web applications.

2 Architecture overview

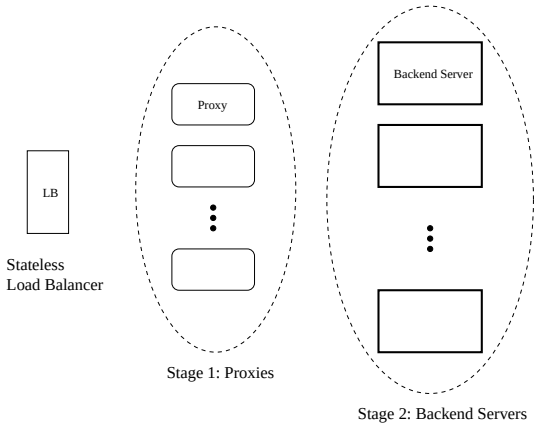

Figure 1. Functional components of our architecture.

Figure 1 shows the functional components of our architecture, which consists of two main stages – a proxy stage and a backend server stage. A stateless load balancer (LB) precedes the proxy stage. This LB distributes the packets received among a cluster of proxy machines. Following the proxies is the backend server stage where the client requests are processed and responses generated.

The LB is stateless and uses a simple load balancing algorithm to evenly spread the load among the proxy machines. It does not modify the client packets in any way. The fact that the LB is completely stateless is a significant advantage of this design, since it makes failure recovery quick and trivial. A primary backup fault tolerance scheme can be used with virtually no need for synchronization, because of the stateless nature of the LB. Multiple LBs can also be used, if required, however, as shown in [5], a single LB can handle a very high packet rate and is usually not a bottleneck.

The proxy stage first accepts a client's TCP connection and HTTP request, and then performs an L7 or content-aware routing and load balancing to pick a backend server suitable for handling that request. Content-aware routing has been extensively studied [8, 9], and its main advantages are: (1) Convenient partitioning of web server content among the servers leading to storage efficiency and scalability; (2) Consolidation of specialized content on specific servers, e.g. one set of servers for video content; and (3) Better performance by exploiting cache affinity [24].

It should be noted that our architecture supports both systems, one where a separate physical proxy is required and the other where it is not. A separate pool of proxy machines can be used if such is required for other reasons such as security. Otherwise, a machine can serve both as a proxy and a server. Although, in the rest of the paper, we talk about proxy machines and backend servers, it should be noted that this distinction is purely on a functional basis. Indeed, a proxy and a backend server may be implemented on the same physical machine. An advantage of not dividing up the machines into proxies and backend servers is that then we do not need to determine the ratio in which they should be divided, which is typically dynamic and not trivial to arrive at. An incorrect partitioning of the servers could lead to the proxy machines or the backend servers becoming a bottleneck.

The proxy, after choosing a backend server, splices the client TCP connection with the connection to the backend

server. There are two issues we need to address at this point: (1) The TCP splice is a single point of failure; and (2) The server response must pass through the proxy, making it a performance bottleneck.

To address the first issue, we replicate the state information required to perform the splicing to all other proxies. This enables any proxy to splice any already spliced connection. In other words, a particular spliced connection is not bound to any particular proxy. This design results in three key advantages: (1) The TCP splice is distributed, that is, subsequent segments of the same spliced connection can pass through different proxies. (2) The TCP splice is fault-tolerant to the failure of a particular proxy machine. In fact, if a proxy machine fails, no explicit recovery action needs to be taken, other than the detection of the proxy failure by the LB, so that future packets are not sent to the failed proxy. (3) By replicating a TCP spliced connection, the server response can now pass through any of the proxy machines, thus eliminating/reducing the above-mentioned performance bottleneck.

In the backend server stage, client requests are processed and appropriate responses are generated. The response is sent to a proxy where it is spliced and sent to the client. An advantage of using TCP splicing is that no changes at all are required at the backend servers, which could be running any OS. However, having to send the response packets through a proxy does limit the scalability of the system. As mentioned earlier, replicating a TCP spliced connection does eliminate/reduce the above-mentioned performance bottleneck. For enhanced scalability, we modify the splicing functionality, so that the architecture has the flexibility of directly sending the response to a client without passing through a proxy.

A TCP splice can be viewed as a combination of two uni-directional splices: a client-to-server splice and a server-to-client splice. The client-to-server splice handles header manipulation of TCP segments from a client to a server, while the server-to-client splice handles header manipulation of TCP segments from a server to a client. At present, both of these unidirectional splices are implemented as a single module on the same machine. We relax this requirement of co-locating the two uni-directional splices on the same machine. In particular, we implement the client-to-server splice in the proxy and the server-to-client splice on the server. Thus, while the proxies splice TCP segments from the client to the backend server, the backend server itself performs the splice of the response segments and sends them directly to the client.

This idea of splitting a splice into two splices is similar in spirit to the splitting of a TCP connections into two pipes as proposed in [17]. However, unlike [17], our proposal does not require any complex changes in the existing TCP/IP stack, nor does it require a new protocol like split-stack in [17] to synchronize the two splices. Splitting the splice functionality does require some changes to the OS of a backend server. However, in cases where such changes are not possible for some backend servers, the architecture supports mixed configurations, that is, some backend servers support a split splice while others do not.

Although the first request on a client connection is L7-routed by a proxy to a backend server, subsequent ones on the same connection are routed by a backend server. If a new backend server is more appropriate of the request, the proxies are informed so that they can “resplice” the connection to the new backend server.

3 Flexible Server Configurations

A key guiding principle of our proposed architecture is that the server components are inexpensive and available off-the-shelf. In fact, our proposed architecture provides a flexibility of merging the proxy and back-end server functionalities on the same set of machines, not changing the OS of back-end servers at all, or even a mixed configuration where the OS of some back-end servers is modified, while no changes are made to the OS of other back-end servers. Figure 2 illustrates three server configurations.

In the first configuration, the proxy functionality is implemented on dedicated machines, and no changes in OS are done in the back-end servers. This architecture is suitable for organizations that require separate proxy machines for reasons, such as, security, and the web service application requires bulk data transfer from client to server, e.g. a repository server for user videos and still photos (similar to Flickr [12]). In the second configuration, the proxy functionality and back-end server functionality are implemented on the same set of machines, and

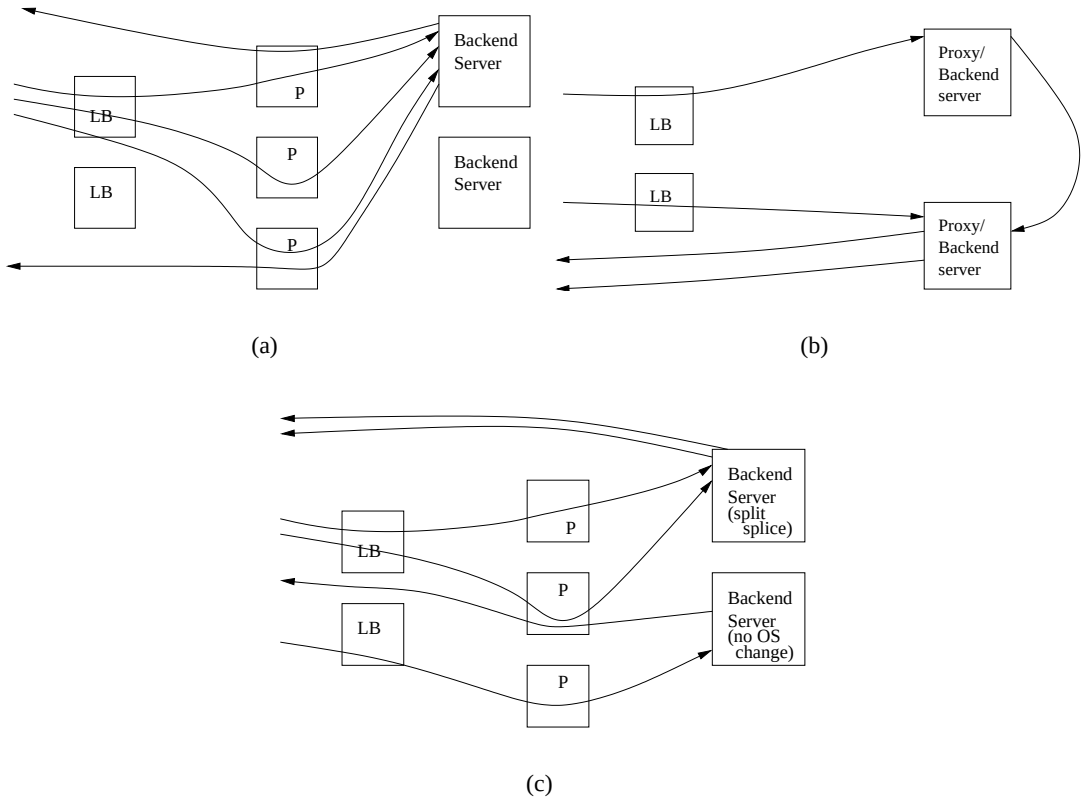

Figure 2. Some examples of possible server configurations. (a) Separate proxy machines; no backend server changes required. The figure shows the paths of packets belonging to a single connection. Packets in both directions need to pass through a proxy, but it could be any proxy. (b) Proxy and backend server on the same machines. A single connection is shown. Here the response packets are always directly sent to the clients. (c) Mixed configuration with separate proxies and split splice installed on some backend servers. The figure shows two connections.

OS changes are required on these machines. This configuration is suitable for organizations that do not require separate proxy machines, and the web service application is traditional, i.e. small client request followed by bulk data transfer from server to client. Finally, the third configuration is a mixed configuration. There are a small number of dedicated proxy machines, as per required by organization for security, and the OS of only some of the back-end servers is changed. The OS of back-end servers providing traditional web service applications is changed (split splice) to have the bulk data flow directly to the clients, bypassing the proxies. On the other hand, the OS of the back-end servers that do not send bulk data to the clients remains unchanged.

4 Load Balancing

Load balancing aims to uniformly distribute client requests among a group of servers, each of which are equally suitable for handling the requests. Ideally, requests should always be sent to the least loaded of the eligible servers. In our architecture, there are two instances where load balancing is required: (1) when incoming client packets are distributed among the proxies, and, (2) when a proxy selects a back-end server for handling a client request. For (1), there is a choice of using a L2 load balancing technique like channel bonding, or, to use a L3 load balancer.

In either case though the packets are not modified in any way. For (2), a proxy acts as a L7 load-balancer.

4.1 Load Balancing Using Channel Bonding

An L2 switch that implements channel bonding can be used to implement load balancing. Channel bonding (also known as trunking or link aggregation or 802.3ad) can be used to load balance at line speeds. Conceptually, channel bonding allows a group of links/interfaces to be treated as one virtual entity. Two major plus points of channel bonding are load balancing and reliability. It load-balances among the links using a user specified algorithm. and also provides inherent, automatic redundancy on point-to-point links. So, should one of the multiple ports used in a link fail, network traffic is dynamically redirected to flow across the remaining good ports in the link. This redirection is fast and the network continues to operate with virtually no interruption in service.

In order to load balance client packet to the proxies, all the ports on an L2 switch connected to the proxies are bonded, as shown in Figure 3.

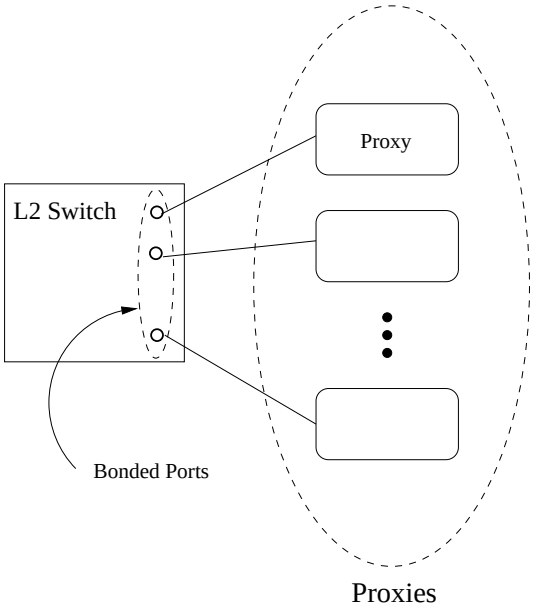

Figure 3. Channel bonding used for ports on an L2 switch.

4.2 L3 Load Balancer

There is an extensive body of research on L3/L4 load balancing. [8, 9] provide a survey of load balancing techniques used in web server architectures. In our prototype implementation, we have used this technique instead of channel bonding, because we did not have access to an L2 switch implementing channel bonding. We use a technique similar to that used in IBM's NetDispatcher [16], with a key difference – our LB does not keep any connection state information. In NetDispatcher and most other similar systems, the LB selects a server for a new TCP connection. Information regarding the connection is hashed and all subsequent packets belong to the same connection are sent to the same server.

Our load balancer is shown in Figure 4. It is assigned a service IP address, which is the address that is advertised to the clients. All the proxies also have the service IP as one of their IP addresses.

A heart-beat (HB) mechanism exists between the LB and each of the proxy machines. This mechanism serves two purposes: (1) it allows the LB to know if a particular proxy has failed, and, if that is the case, to stop sending

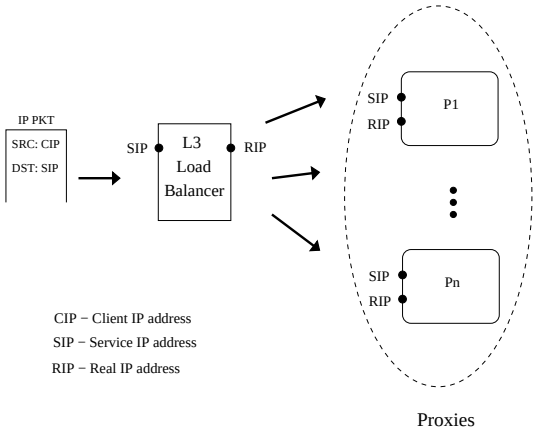

Figure 4. A layer 3 load balancer.

packets to it; (2) as part of the HB, a proxy sends an "workload" factor which reflects the resources currently available on that proxy.

The LB uses a load balancing algorithm (described later) to determine which proxy to send the next packet to. Note that since the proxies also have the service IP address as a virtual address, no changes are required to the IP packet header; the packet is simply sent to the selected proxy by using the proxy's MAC address.

Using the above mechanism for load balancing requires that all the proxies be in the same subnet. This requirement can be weakened by the use of IP tunneling. However, this results in some additional overhead.

The LB is not a single point of failure as a backup LB can be used. The important point to note here is that the LB does not have any hard state; therefore no state information needs to be exchanged with a backup and failover to a backup in the event of a failure is simple and fast.

Our load-balancing algorithm is based on [7]. Consider a load balancer, L, and N servers. In order to balance load among the N servers, L maintains a heart-beat (HB) mechanism with each of the servers. L receives a workload factor, W_i, from the ith server every HB interval. These workload factors represent the amount of available capacity at a server and are determined based on factors, such as, number of existing connections, CPU and memory usage. Note that these factors are actually fractions of the maximum capacity, and, therefore, the actual capacity is not relevant and can be different across servers. The algorithm tries to schedule requests such that each server receives requests proportional to its workload factor. A convenient way to do this is by normalizing the W_i's: $W_i' = \frac{W_i}{\sum_1^N W_i}$. Then, to schedule a request, a random number, r, is generated and the request is scheduled to server, s_i, if $r \in [\sum_1^{i-1} W_j', \sum_1^i W_j']$.

5 Proxy Architecture

By virtue of distributing and replicating spliced TCP connections among all proxy machines, any proxy machine can accept a client's connection and subsequent requests from that client may flow through any of the proxy machines. However, to create a new TCP splice (for a new client server session), the first client request that is used to do layer-7 routing must be processed by the same proxy machine that accepted the initial TCP connection. Notice that a TCP splice is created only after receiving this first client request. To ensure the distribution of the load of creating spliced connection evenly, we divide the set of proxy machines in multiple groups. A new client connection is identified by a hash computed on source IP address, destination IP address, source port number and destination port number. We provide a mapping between this hash and a proxy group that creates the corresponding

spliced connection. Figure 5 shows the sequence of steps involved in handling a client request. These steps are described below:

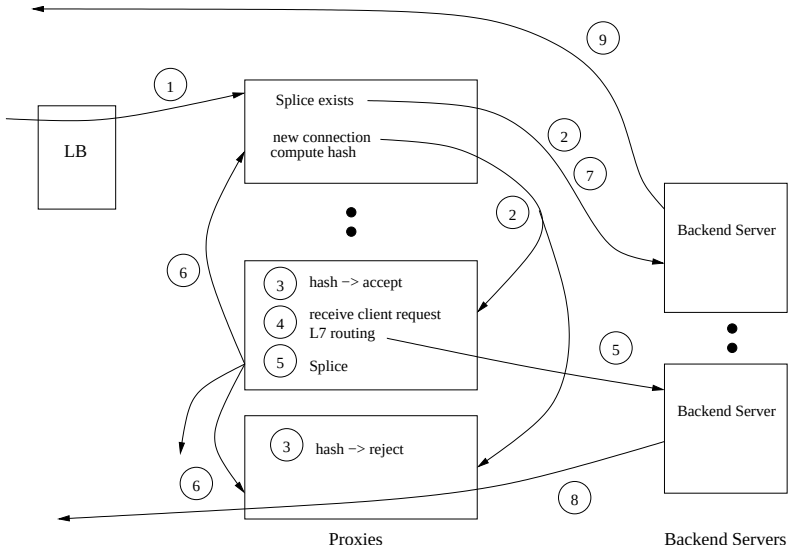

Figure 5. Sequence of steps involved in handling a client request.

1. The LB receives a client packet and uses its load balancing algorithm to forward it to a proxy.
2. The proxy receiving the TCP segment determines if it is a new client connection. It does this by looking at the IP addresses and port numbers in the segment and comparing them to its TCP splice state information. If the segment matches an existing splice, it is spliced and sent off to the corresponding backend server. Otherwise, a hash is computed for it, and the proxy group that is assigned for creating its splices is determined. The the segment is then multicast to the members of that proxy group.
3. The proxies in the proxy group receive the segments. Here another hash is computed and precisely one member of the proxy group accepts the segment.
4. That proxy accepts the client TCP connection and waits for the client request to arrive. Once it has the client request, it uses its L7 routing algorithm to find an appropriate backend server.
5. The proxy opens a TCP connection with the chosen backend server, sends it the client request and splices the two TCP connections.
6. The proxy then sends the splicing state information of this connection to all other proxies.
7. Further segments from the client arriving on that TCP connection (e.g. ack segments) can now be spliced by any proxy to the backend server chosen earlier.
8. The backend server on receiving the request, processes it and sends the response to the client. The server can use any proxy for the response.
9. If split splice is installed on a backend server, the response is directly sent to the client without passing through a proxy.

6 Implementation

We have implemented a prototype of our system in Linux kernel version 2.6.12. This implementation consists of the following major parts: (1) TCP splicing code, including support for distributed splicing and split splicing; (2) Load balancing code; and (3) Proxy code.

In our implementation, we have used Netfilter [22] quite extensively. Netfilter adds a set of hooks along the path of a packet's traversal through the Linux network stack. It allows kernel modules to register callback (CB) functions at these hooks. Five such hooks are provided, as shown in Figure 6. These hooks intercept packets and invoke any CB functions that may be registered with that hook. When multiple CB functions are registered at a particular hook, they are invoked in the order of the priority specified at the time of their registration. After processing a packet, a CB function can decide to inject it back along its regular path, or steal it from the stack and send it elsewhere, or even drop it.

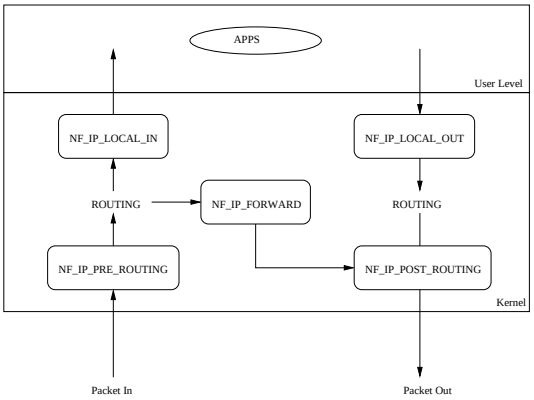

Figure 6. Netfilter allows functions to be registered at five different hooks in the network stack.

6.1 Load Balancer

A software based load-balancer (LB) is used for the prototype. For better performance, a HW based load balancer could be used. However, since we are more interested in comparing performance rather than measuring the absolute performance of the system, a SW based LB is equally suitable for our experiments. Our LB, which is a Linux kernel module, is based on code from the Linux Virtual Server (LVS) project [18]. However, one key difference and advantage of our LB is that it is stateless, whereas LVS keeps a hash table of connections so that all packets of a connection are sent to the same server. The NF_IP_LOCAL_IN netfilter hook is used for intercepting packets. In the network stack, this is right after the IP packets have been defragmented. (This has the added advantage that the proxies do not have to worry about fragmented IP packets.)

The packets received for the service IP address are load-balanced among the MAC addresses of the proxies. Each of the proxy has a service IP address and a "real" IP address. These real IP addresses are configured at the load balancer. Using the load-balancing algorithm, the LB chooses a real server to which to deliver a packet, and then sends it out. The difference here from regular routing is that the packet is sent to the MAC of the proxy's "real" IP address instead of the packet's destination address.

However, for the experiments that follow, the LB simply uses a round-robin mechanism to distribute the load to the proxies. It sends 40 packets to a particular proxy before switching to the next one.

6.2 Proxy

The modifications made to the proxies can be divided into kernel and user level implementation.

Kernel level implementation. The kernel level changes at the proxies are implemented as Linux kernel modules called `tcpdistsplice` and `packetredirector`.

1. `tcpdistsplice` This module provides the TCP splicing and distributed splicing functionality. It is based on the TCP splicing module available at the Linux Virtual Server (LVS) project [29]. It registers with the NF_IP_LOCAL_IN netfilter hook to intercept any arriving IP packets.

   Netfilter also allows socket options and handlers to be defined. These are used for communication between a user process and `tcpdistsplice`. `tcpdistsplice` defines three socket options — `PRE_SPLICE, SPLICE, DIST_SPLICE`. In order to establish a splice between two TCP connections, a user process needs to call `setsockopt()` with `PRE_SPLICE`, and then with `SPLICE`. After receiving a `PRE_SPLICE`, the module drops and does not acknowledge any further packets from the client until the splice is established. The module also assumes that the server does not send any data after connection establishment until the client request is received. These restrictions can be relaxed, and, the splicing module made more efficient [27], however, that is not a focus of this paper.

   `DIST_SPLICE` is used for splicing on a remote machine. These splices are created based on state information, without the presence of the corresponding TCP level sockets. The `setsockopt()` call with `PRE_SPLICE` returns state information such as TCP sequence numbers related to the splice. This information is then communicated to other proxies to use with `DIST_SPLICE`.

2. `packetredirector` This module ensures that all the client packets from a particular connection are handled by the same proxy before a TCP splice for that connection is created. It also uses the NF_IP_LOCAL_IN netfilter hook, but with a lower priority than the TCP splicing module, that is, `tcpdistsplice` is always called first on a packet. If there is no match, that is, there is no splice for that connection, `tcpdistsplice` re-injects the packet back into the stack and `packetredirector` is called on it.

   This module redirects packets based on computation of a hash. The hash is based on the source and destination IP addresses and port numbers. Note that packets that the LB sends to the proxies are already defragmented. After a TCP splice is created for a particular connection, and distributed to other proxies, packets belonging to that connection find a match in the splicing module and are never sent to the redirection module.

User level implementation. A user level proxy that accepts client connections, connects to a server, passes splicing state information to other proxies, splices the client and server TCP connections was implemented. The significant parts of the proxy implementation are shown in Figure 7.

A process called `spliceListener` also runs on a proxy. This process waits to receive splicing state information from other proxies, and then uses that information to create a `DIST_SPLICE`.

6.3 Backend Server

There is an optional modification that is made to the backend servers. For greater scalability, the splicing functionality can be split between the proxies and the backend servers. We modify the `tcpdistsplice` module and register it at the NF_IP_LOCAL_OUT netfilter hook to implement this functionality. This hook is invoked right after a local process has sent out an IP packet. Other than the fact that it is registered at a different hook, this module is identical to `tcpdistsplice`.

```
1 c1 = accept() // client connection
2 n = read(c1) // client request
3 s = L7routing() // determine server
4 c2 = connect(s) // to server

// do pre-splice
5 setsockopt(..PRE_SPLICE,c1,c2,info..)

// read any additional client data
6 n += read(c1) // (non-blocking)

//send splice info to other proxies
7 sendProxies(info, n)

// set up splice
8 setsockopt(..SPLICE,c1,c2,n..)

9 write(c2) // n bytes to server
```

Figure 7. Pseudocode of the proxy implementation.

The mechanism of sending the splice state information to a backend server and that of establishing a split splice there is identical to that of establishing a distributed splice at a proxy. Indeed, the functionality of a split splice at a backend server is identical to that of a proxy performing a distributed splice with the only difference being that the server split splice module will always receive packets in only one direction.

7 Experimental results

The goal of our experiments is to demonstrate fault-tolerance and scalability of our architecture. Our objective is not to test the capacity of a server. In particular, we perform three sets of experiments with a focus on (1) proxy scalability, (2) proxy failover, and (3) split splice architecture.

7.1 Experimental Setup

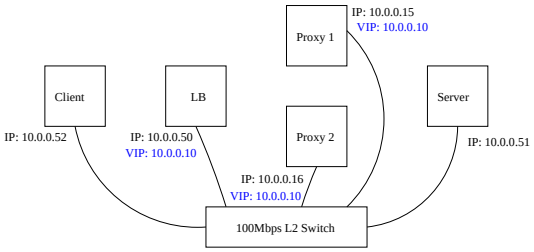

Figure 8. Experimental setup.

We use five machines for our experiments. Specific details of the machines are listed in Table 1. The experimental setup is shown in Figure 8. The machines are connected together using a 5 port 100 Mbps Linksys Ethernet Switch.

Client:	HP DX2000, CPU 2.66 GHz, RAM 128MB, Ethernet 100 Mbps
Load Balancer:	HP DX2000, CPU 2.66 GHz, RAM 128MB, Ethernet 100 Mbps
Proxy 1:	Dell Inspiron 9300, CPU 1.6 GHz, RAM 512MB, Ethernet 100 Mbps
Proxy 2:	Dell Inspiron 6000, CPU 1.5 GHz, RAM 512MB, Ethernet 100 Mbps
Server:	HP DX2000, CPU 2.66 GHz, RAM 128MB, Ethernet 100 Mbps

Table 1. Specifications of the machines used in the experiments.

The client sends requests to the virtual service address (10.0.0.10). Since the load balancer and both the proxies are on the same LAN in our setup, this can lead to a conflict when the client sends ARP to determine the corresponding MAC address. There are various ways to deal with this issue [6], for example, by making sure that only the LB responds to the ARP requests for the VIP, and the proxies ignore those requests. Here, we have chosen a simple solution of creating a static ARP entry on the client, mapping the service VIP (10.0.0.10) to the MAC address of the LB.

In a real system, if separate proxy machines are used as proxies, they will typically be multi-homed with separate interfaces for the server and the clients for security reasons. However, that is not a concern in our experiments.

7.2 Proxy scalability

In these experiments we measure the time taken to transfer files of different sizes from clients to a server using the HTTP PUT method. 100 simultaneous transfers are done in each case. A `C` program was written to simulate 100 clients. It calls `fork()` 100 times and each of the 100 children connect to the server and sent the PUT request followed by the data.

We chose three files sizes: 100 KB, 10 MB, and 20 MB. The reason behind these choices is that these could represent the sizes of music, pictures, and videos that a user may upload to a server through her web browser. For each of these file sizes, we conduct experiments for three different configurations: (1) C–S: clients connect directly to the server. We use this configuration as a base case to compare with the other cases. (2) C–LB–P–S: There is an L3 load balancer (Section 4.2) and a proxy (Section 5) between the client and the back-end server. (3) C–LB–2P-S: There are one load balancer and two proxies between the client and the back-end server.

Since we are interested in measuring the scalability of the proxies, we monitored the CPU and memory usage during each of the experiments on all the machines to make sure that the client, server, or the load balancer were not becoming a bottleneck. `sar` was used to make these CPU and memory usage measurements. We synchronized the clocks on all the machines. Furthermore, the client logged the start and the end time of the experiments. We then used these times to obtain the CPU and memory load on all the machines.

We took at least three measurements for each experiment, excluding the first run which was discarded to minimize the effect of cache misses.

An Apache [3] web server was run on the server machine. A simple CGI script was written to handle the PUT requests. It looks at the “Content-length” tag in the HTTP request header, reads those many bytes, and discards them.

7.2.1 Discussion of results

The results are summarized in Tables 2, 3 and 4 for transfer sizes 20 MB, 10 MB and 100 KB, respectively. The average time in these tables is the average for a single connection, and, the total time is the duration of the experiment, from the time the first connection starts to the time when the last one ends.

Tables 2 and 3 also show the percentages of CPU idle time and memory used. The CPU idle time was 98 to

100% for all machines before the experiments were run. The results show that none of the machines were CPU or memory bound during the experiments. This implies that the data transfer rate is network bound.

We observe that the average and total times of C–LB–2P–S configuration are almost same as those for the base configuration (C–S). However, the one proxy configuration is 6.46% and 8.8% slower for the 20 MB and 10 MB case, respectively. We make a similar observation from the net I/O throughput (Total number of bits transferred per second to the server) achieved from these experiments. The net I/O throughputs are almost same in the base case and the C–LB–2P–S configuration. However, it is about 5.89% and 5.5% lower in the one proxy configuration.

The CPU load on proxy P1 is about 25% due to TCP splicing in the one proxy configuration . We see that this load decreases by about 7% in the two proxy configuration. Another thing to note is that the CPU load on the proxies is the same for both 10 MB and 20 MB transfers.

While these experiments show some improvements by introducing another proxy machine, it is important to note that the system is still network bound. It would be interesting to repeat these experiments with gigabit Ethernet switch and interfaces. We expect that the proxy will become CPU bound in that case. This is based on the fact that the load on the proxy P1 increases by about 25% during the experiments. If the network speed increases by a factor of 10, it is quite likely the transfer rate will become CPU bound. In that situation, introducing additional proxies would lead to much higher performance improvement than what we see here.

We want to emphasize the scalability advantage of a distributed TCP splice over a non-distributed TCP splice. If the traffic on an existing persistent TCP connection that is traditionally spliced (non distributed) through a proxy increases, there is no way to spread the increased load to other proxies, even if other proxies are present. With distributed splice, however, workload of existing TCP connections can be spread across other proxies.

Config.	Ave. Time (s) (std. dev.)	Total time	Net I/O (Mbps)	Client		LB		P1		P2		Server	
				CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM
C–S	165.02 (7.20)	177.80	94.36	93	96	n/a	n/a	n/a	n/a	n/a	n/a	74	95
C–LB–P–S	175.69 (8.67)	188.85	88.81	93	96	89	64	74	74	n/a	n/a	66	97
C–LB–2P–S	165.83 (8.52)	177.83	94.34	93	96	89	66	81	77	66	95	73	88

Table 2. Experimental results for client doing 100 concurrent PUT requests of a file of size 20 MB

Config.	Ave. Time (s) (std. dev.)	Total time	Net I/O (Mbps)	Client		LB		P1		P2		Server	
				CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM
C–S	78.99 (8.27)	89.40	93.83	93	95	n/a	n/a	n/a	n/a	n/a	n/a	74	95
C–LB–P–S	85.97 (6.09)	94.59	88.67	93	95	89	61	74	92	n/a	n/a	76	95
C–LB–2P–S	79.78 (8.32)	89.42	93.81	93	95	90	68	81	73	66	96	74	96

Table 3. Experimental results for client doing 100 concurrent PUT requests of a file of size 10 MB

Config.	Ave. Time (s) (std. dev.)	Total time	Net I/O (Mbps)
C–S	0.738 (0.124)	1.87	43.81
C–LB–P–S	0.784 (0.463)	4.57	17.93
C–LB–2P–S	0.723 (0.12)	1.87	43.81

Table 4. Experimental results for client doing 100 concurrent PUT requests of a file of size 100 KB

7.3 Proxy Failovers

The objective of these experiments is to show that the impact of a proxy failure on the users is negligible. In these experiments, the client sends 100 simultaneous PUT requests to the server, similar to the experiments in the previous section.

Each request transfers 10 MB to the server. Experiments are performed with a two proxy configuration. A proxy is failed during the transfer by disabling its Ethernet interface. This failure is detected by the load balancer and it subsequently stops sending packets to that proxy. The failure detection time depends on the frequency of the heartbeat (HB). We conducted experiments with three different values of the HB frequency: 5s, 1s and 200ms. UDP is used for the HB, and the LB decides that a proxy has failed if it misses three HB in a row. In that case, it remove the failed proxy from its list of active proxies. Thus, for a HB interval of t sec., a failure is detected in between $2t$ and $3t$ sec.

Since after a failover, only one proxy is available for the remaining data transfer, the total time taken also depends on when the failure occurs after the start of the transfer. If the failure occurs right after the transfer starts, then the total time taken is expected to be closer to that of the one proxy configuration (C–LB–P–S). On the other hand, if the failure occurs towards the end of the transfer, the total time taken is likely to be closer to that for the two proxy configuration (C–LB–2P–S). Thus, to remove this additional variable, and to be able to meaningfully compare experiments with different HB frequency, we fail a proxy after half of the average non-failure transfer time in these experiments. For 10 MB, since the non-failure case takes about 90 sec, we failed a proxy at about 45 sec. after the start.

7.3.1 Discussion of results

	Time (in sec.)					Network
HB Interval	Avg.	Min	Max	Std. dev	Total	Bandwidth (Mbps)
no failure	79.76	46.35	89.37	8.18	89.46	93.76
5 sec	90.54	54.66	142.07	8.86	142.14	59.02
1 sec	85.69	55.88	97.88	5.51	97.95	85.64
200 ms	82.20	45.30	94.42	5.70	94.48	88.79

Table 5. Experimental results for concurrent PUT of 10 MB by 100 client connections with a proxy failure during the transfer.

The results are summarized in Table 5. As expected, the average transfer times are longer for larger HB intervals. If the failures were to be detected instantaneously, the time taken should approximately be an average of the time taken with the C–LB–P–S and C–LB–2P–S configurations, which is 82.87 sec. (from Table 3). For 200 ms HB (for which failure is detected in 400 to 600 ms), the average time taken is almost exactly this (in fact, it is slightly lower, which can be attributed to some inaccuracy in when the failure was caused).

For 1 and 5 sec HB intervals, it takes 2.5 and 12.5 sec on average to detect failure. During this time half the packets sent out by the client are dropped. This would cause the client to invoke TCP congestion control.

We have also calculated the net I/O throughput achieved in each of these cases. The net throughput of course decreases when there is a proxy failure. This is because of time spent on failover. However, we do observe that this throughput remains relatively large when the HB interval is short.

7.4 TCP Split Splice

The objective of these experiments is to show the advantage of TCP split splice, where the server performs the TCP splice header transformations on the response packets and sends them directly to the client.

As before, we again creates 100 concurrent connections. However, we used the HTTP GET method to download files from the server in these experiments. Similar to the earlier experiments, we transfer files of two different sizes: 10 MB and 20 MB.

We use three different configurations. (1) C–S, where the client directly connects to the server; (2) C–LB–P–S, where one proxy is used; and (3) C–LB–P–S*, which is similar to the previous configuration with the difference that split splice kernel module is installed on the server.

7.4.1 Discussion of results

	Time (in sec.)					Network
Config.	Avg.	Min	Max	Std. dev	Total	Bandwidth (Mbps)
C–S	78.93	49.76	88.33	5.35	88.40	94.89
C–LB–P–S	84.91	55.13	93.58	4.78	93.66	89.53
C–LB–P–S*	83.18	62.35	90.39	3.93	90.45	92.69

Table 6. Experimental results for concurrent GET of 10 MB by 100 client connections with split splice at the server.

	Time (in sec.)					Network
Config.	Avg.	Min	Max	Std. dev	Total	Bandwidth (Mbps)
C–S	165.76	132.07	176.75	5.87	176.79	94.78
C–LB–P–S	174.12	133.38	186.27	6.63	186.36	90.01
C–LB–P–S*	170.79	144.63	180.39	5.26	180.09	93.21

Table 7. Experimental results for concurrent GET of 20 MB by 100 client connections with split splice at the server.

Tables 7 and 6 show the performance measured for 20 MB and 10 MB files respectively. We notice that the average and total times of C–LB–P–S* configuration are higher than those in the base case, but lower than those in the C–LB–P–S configuration. So, while we do see some performance improvements due to split splice, it is not as substantial. We expect that with larger file sizes, we will see more performance improvements. Furthermore, all component (client, LB, proxies and server) were connected via a single switch in our experiments. The key advantage of split splice architecture is when there are several alternate paths from the server the clients. We expect a much large performance improvement if there are multiple switches interconnecting the four components.

8 Scaling to Very Large Server Farms

Web services organizations like Google and Yahoo have server farms consisting of tens of thousands of machines. Servers can be partitioned by the service they provide – for instance, a separate cluster could be used for

each of the services that Google or Yahoo provide. However, even for a single service, like search or email, the number of servers required can run from thousands to a few tens of thousands.

Although the architecture described above scales well, ultimately, with the increase in the number of the machines and the client connections per second, sharing of all the splice state information will all the proxies is likely to be an issue. We address this by organizing the proxies into sub-clusters, with the size of each sub-cluster in the order of a hundred to a few hundred machines. The splicing state information is distributed only within a sub-cluster.

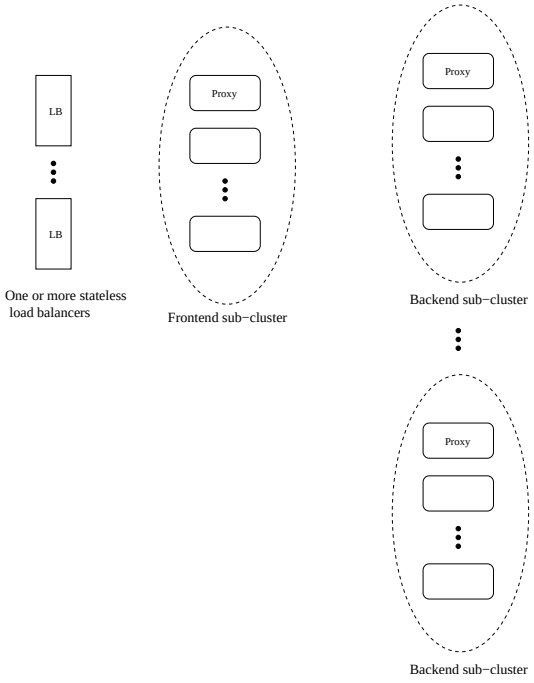

Figure 9. Machines are organized into sub-cluster for a large web server farm.

The sub-clusters are organized in two levels. There is one front-end sub-cluster and many backend sub-clusters. This is shown in Figure 9. The load balancers distribute the client packets among the machines in the front-end sub-cluster. These machines first check if a splice already exists for a packet. If it does, the splice is performed and the packet sent out. Otherwise, a hash, based on IP addresses and port numbers, is computed to determine a backend sub-cluster. Once a sub-cluster is determined, the actual machine in that sub-cluster is chosen based on a load-balancing mechanism similar to that used by a front-end load-balancer. Each backend sub-cluster is identical to the proxy/server architecture described in the previous sections.

Assuming a sub-cluster of about 100 machines, and about 100 sub-clusters, results in a server farm of about 10K machines. It is assumed here that a physical machine acts as both a proxy and a backend server.

The pseudo code for the logic implemented in the front-end sub-cluster for handling a client packet is shown in Figure 10.

9 Related Work

The design of server architectures has been an active area of research for the past several years. The main focus of this research has been on enhancing this typical server architecture to make it highly scalable, responsive, reliable and cost effective. A good survey of some of the earlier web server architectures and the related issues is

```
if (spliceExists(pkt))
    spliceItToBackendServer(pkt)
else { //splice does not exist
     //in the FE sub-cluster
    sub_cluster = hash(pkt)
    if (sub_cluster == own) {
        group = hash1(pkt)
        multicast(pkt, group)
    } else {
        proxy = LB(sub_cluster)
        send(pkt, proxy)
    }
}
```

Figure 10. Pseudocode for logic implemented in the frontend sub-cluster proxies.

given in [8]. Content-based routing using a layer-7 switch allows the front-end to parse in-coming requests and make a decision about which back-end server to dispatch the request to [10, 2]. Research has shown that content-based routing significantly improves the scalability of web servers. TCP splicing[19, 20] and TCP handoff [24] mechanisms were introduced to optimize content-based routing approach.

Based on where the functionalities of receiving client requests, parsing client requests and forwarding them to the appropriate back-end server (content-based routing), we can classify current server architectures into three categories: front-end based systems, back-end based systems and hybrid systems. All three functionalities are implemented in the front end in front-end based systems, in the back end in back-end based systems, and split among front end and back end in the hybrid systems. Advantages of front-end based systems include (i) cluster management policies and administration are encapsulated in a single machine; and (ii) back-end servers do not require any changes and hence can be unaware of the cluster. The disadvantages are (i) low scalability, since front-end is a bottle-neck; and (ii) front-end is not fault-tolerant. Examples of front-end based systems are [10, 16].

Advantages of back-end based systems include high scalability and server fault tolerance. The main disadvantage is that they require modifications to the OS of the back-end servers. Examples of back-end based systems are [17, 5, 28]. Finally, the advantages of hybrid systems are high scalability, server fault tolerance, and an ability to efficiently manage subsequent requests from the same clients. The main disadvantage is that they require modifications to the OS of the back-end servers. Examples of hybrid systems are [15, 4].

10 Conclusions and Future Work

A quiet revolution is taking place in the way we use the Internet. User applications that usually run on a machine locally are beginning to be offered remotely through a web browser for ease of both accessibility from any location and sharing information with others. Furthermore, users are increasingly using servers to store large data, such as images and videos. There are two distinct features of these new types of applications that are not present in the traditional web services. The newer applications require relatively long-duration, stateful client server sessions, and the amount of data flowing from the clients to the servers is relatively large.

This paper proposes a server architecture that addresses the fault tolerance needs of such applications. In particular, three enhancements to TCP splicing mechanism are proposed, and a prototype of a flexible web server architecture based on these enhancements is built. Performance measured from this prototype is encouraging. It supports the key objectives that the proposed architecture is scalable and fault tolerant.

There are three areas of future work that we are pursuing. First, we are extending our experiments to include multiple gigabit Ethernet switch and interfaces. As discussed in Section 7, this is expected to make the proxies

CPU bound and provide further performance improvements in the case of split splicing.

The second area of future work we are pursuing is that there are two features of the proposed architecture that we have not yet implemented. Note that both of these features do not have any impact on the performance results we have presented in the paper. The first feature we haven't yet implemented is re-splicing of a TCP connection to a different backend server. This may be required when multiple requests of different types are received on the same HTTP connection. This implementation should be straightforward, very similar to how re-splicing is implemented in KNITS [15]. The second feature we haven't yet implemented is a more sophisticated load balancing algorithm in the LB. In the current prototype, the load balancer uses round-robin to distribute the load among the proxies, rather than considering the workload on the proxies as described in Section 4.2.

Finally, the third area of future work we are pursuing is dealing with backend server failures. The current prototype tolerates proxy failures, but not backend server failures. While replicating data on multiple servers is a straightforward solution that is being used by many web service providers, our goal is to address the issue of failure of computing servers such that it is completely seamless to the user. With the emergence of long-duration, stateful client server sessions, providing client-transparent, seamless service to the clients in the face of back-end server failures is non-trivial. One important issue that we need to deal with is non-deterministic nature of most server applications.

References

[1] Ajax, http://wikipedia.org/ajax.

[2] Alteon websystems, http://www.alteonwebsystems.com.

[3] Apache. http://apache.org.

[4] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for P-HTTP in cluster-based web servers. In *Proceedings of USENIX'99*, 1999.

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

[6] ARP Issue, http://www.linuxvirtualserver.org/docs/arp.html.

[7] J. Aweya, M. Ouellette, D. Y. Montuno, B. Doray, and K. Felske. An adaptive load balancing scheme for web servers. *Int. Journal of Network Management*, 12(1):3–39, 2002.

[8] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. *ACM Comput. Surv.*, 34(2):263–311, 2002.

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

[10] Cisco content services switches, http://www.cisco.com.

[11] A. Cohen, S. Rangarajan, and J. H. Slye. On the performance of tcp splicing for url-aware redirection. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[12] Yahoo's flickr, http://flickr.com.

[13] Google mail, http://gmail.com.

[14] Google maps, http://google.com/maps.

[15] E. V. Hensbergen and A. E. Papathanasiou. KNITS: Switch-based connection hand-off. In *IEEE Infocom*, 2002.

[16] Ibm interactive network dispatcher, http://www.ibm.com/dispatcher.

[17] R. Kokku, R. Rajamony, L. Alvisi, and H. Vin. Half-pipe anchoring: An efficient technique for multiple connection handoff. In *Proceedings of ICNP*, Paris, France, Nov 2002.

[18] LVS project, http://www.linuxvirtualserver.org.

[19] D. Maltz and P. Bhagwat. TCP splicing for application layer proxy performance, 1998.

[20] D. A. Maltz and P. Bhagwat. MSOCKS: An architecture for transport layer mobility. In *Proceedings of INFOCOMM'98*, March 1998.

[21] Microsoft's virtual earth, http://local.live.com.

[22] Netfilter, http://www.netfilter.org.

[23] Num sum, http://numsum.com.

[24] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-aware request distribution in cluster-based network servers. In *ASPLOS*, pages 205–216, 1998.

[25] M.-C. Rosu and D. Rosu. An evaluation of TCP splice benefits in web proxy servers. In *WWW*, pages 13–24, 2002.

[26] M.-C. Rosu and D. Rosu. Kernel support for faster web proxies. In *USENIX Annual Technical Conference, General Track*, pages 225–238, 2003.

[27] O. Spatscheck, J. S. Hansen, J. H. Hartman, and L. L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 8(2):146–157, 2000.

[28] W. Tang, L. Cherkasova, L. Russell, and M. Mutka. Modular tcp handoff design in streams-based tcp/ip implementation. In *Lecture Notes in Computer Science*, volume 2094. Springer-Verlag, 2001.

[29] Linux TCP splicing module, http://www.linuxvirtualserver.org/software/tcpsp.

[30] Writeboard, http://www.writeboard.com.

[31] Writely, http://www.writely.com.

[32] L. Zhao, Y. Luo, L. Bhuyan, and R. Iyer. Design and implementation of a content-aware switch using an network processor. In *13th HOT Interconnect*, 2005.

[33] Zoho writer, http://zohowriter.com.