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 describes three enhancements to the TCP splicing mechanism: (1) Enable a TCP connection to be simultaneously spliced through multiple machines for higher scalability; (2) Make a spliced connection fault-tolerant to proxy failures; and (3) Provide flexibility of splitting a TCP splice 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 measured from this implementation.

1 Introduction

Internet services are commonly offered over the web using application layer proxies. These proxies perform a number of important functions, including layer-7 or content-based routing that routes different client requests to the appropriate application servers based on request content. Other functions performed by these proxies 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.

TCP splicing [20, 27] has been commonly used for improving the performance of serving web content through proxies. It avoids any context switches or data copying between kernel and user space, resulting in improved performance. In fact, it has been shown that TCP splicing makes the performance of a proxy comparable to that of IP forwarding [19]. Advantages of using TCP splicing to build web servers are described in [26, 25, 11].

At present, a TCP splicing based web server architecture suffers from two drawbacks: (1) All traffic between clients and servers (both directions) must pass through a 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 reissue failed requests, even in the presence of a backup proxy.

This paper addresses these two drawbacks of a TCP splicing based web server architecture. In particular, the work presented in this paper consists of two major components. The first component consists of providing three enhancements to the TCP splicing method. These are: (1) design and implementation of a replicated TCP splice; (2) design and implementation of a fault-tolerant TCP splice; and (3) design and implementation of a split splice. These enhancements allow a TCP connection to be spliced at multiple proxies, providing both fault-tolerance and higher scalability. In particular, failure of a proxy causes no disruption to a splice, and additional proxies can be added to scale the system. To further increase the scalability of a web server architecture, a TCP splice can be split between the proxies and a backend server, with the backend server performing the splicing functionality for the response packets. We describe the design, implementation and evaluation of these three enhancements to TCP splice.

The second component of this paper consists of using these TCP splice enhancements to build a flexible, scalable and fault-tolerant web server architecture. This architecture addresses the requirements of the *newly emerging web service applications*. The nature of user services provided over the Internet is changing. User applications that were usually run on a local machine are beginning to be offered remotely through a web browser for ease of both accessibility from any location, and sharing of 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.

These newly emerging web service applications have two distinct features that are not present in traditional web service applications: (1) They maintain relatively long-duration, stateful client-server sessions; and (2) The amount of data flow from the clients to the servers is relatively large. As users become dependent on such services in increasing numbers and use them for critical tasks, a failure of servers hosting such applications can cause significant disruption. For a seamless user experience during server failures, it is not sufficient to simply provide server fault-tolerance, e.g., by using data replication and alternate servers. The *quality* of server fault-tolerance, in terms of the impact of a failure on a user, is also very important. In some cases, it may not be acceptable for the recovery process to take tens of seconds. For instance, this 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, focusing 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 works well for traditional web service applications that mainly download webpages to the client machines, it is unacceptable for newly emerging web service applications that are long-duration, highly interactive, and stateful in nature.

We describe the design, implementation and evaluation of a web server architecture that addresses this limitation. The new architecture is based on the three enhancements to the TCP splicing functionality, and provides enhanced fault-tolerance. 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. It is provided without sacrificing scalability or efficiency. In fact, the new architecture is as scalable and efficient as the earlier web server architectures [5, 17] that do not provide such enhanced fault tolerance support.

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], it is significantly more expensive than a network interface card (NIC). Furthermore, our architecture could be implemented in a network processor leading to a better performance.

The rest of the paper is organized as follows. In the next section, we provide a brief description of TCP splice and discuss some related work. In Section 3, we describe the overall system architecture. This is followed by a description of some architectural configurations. Load balancing and proxy architectures are presented in Sections 5 and 6, respectively. This is followed by sections on our prototype implementation of the system on Linux and experimental results. Finally, the conclusions are presented in Section 9.

2 Background

2.1 TCP Splice

Web proxies are exceedingly used in web server architectures for implementation of layer 7 (or content-aware) routing, security policies, network management policies, usage accounting and web content caching. An application level web proxy is inefficient since relaying data from a client to a server involves transferring data between kernel space and user space, and, the associated context switches. TCP Splice was proposed [20, 27] to enhance the performance of web proxies. It allows a proxy to relay data between a client and a server by manipulating packet header information, which is done entirely in the kernel. This makes the latency and computational cost at a proxy only a few times more expensive than IP forwarding. There is no buffering required at the proxy that performs the splicing, and, furthermore, the end-to-end semantics of a TCP connection are preserved between the client and the server. Advantages of TCP splicing in web server architectures are further described in [26, 25, 11].

The following steps are required in establishing a TCP splice.

- The client connects to a proxy. The proxy accepts the connection and receives the client request.

- It may perform authentication and other functions as configured by the administrator. Then it performs layer 7 routing to select a server and initiates a new TCP connection to it.

- At this point, the proxy sends the client request to the server and "splices" the client–proxy and the proxy–server TCP connections.

- After the two TCP connections are spliced, the proxy acts as a relay — the packets coming from the client are sent on to the server, after appropriate modification of the header (which makes the server believe that those packets are part of the original proxy–server TCP connection); similarly, the packets received from the server are relayed on to the client (after header modifications).

2.2 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 the 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 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 backend 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 client requests to the appropriate backend server (content-based routing) are implemented, we can classify current server architectures into three categories: front-end based systems, backend based systems and hybrid systems. Advantages of front-end based systems include (i) cluster management policies and administration are encapsulated in a single machine; and (ii) backend 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 backend based systems include high scalability and server fault-tolerance. The main disadvantage is that they require modifications to the OS of the backend servers. Examples of backend 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 backend servers. Examples of hybrid systems are [15, 4].

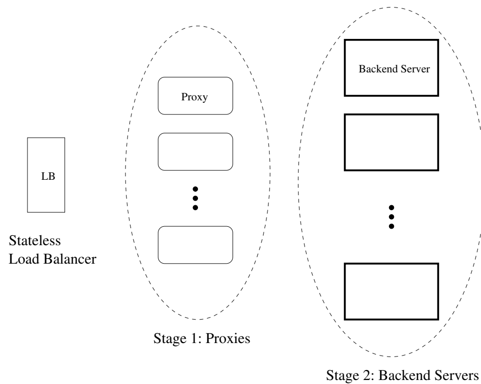

Figure 1: Functional components of our architecture.

3 Architecture overview

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. Note that this is only a functional representation of our architecture and that in an actual implementation some of these components may be co-located, e.g., the LB could be co-resident with a proxy, or, a proxy may be co-resident with a backend server.

The LB works at the IP layer, is stateless and uses a simple load balancing algorithm to evenly spread the load among the proxy machines. It spreads the load at the IP level, and 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 be used for fault-tolerance, 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 another set of servers for audio; 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 made 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 proxy that creates the TCP splice is a single point of failure; and (2) All traffic on the spliced connection (both directions) must pass through the proxy that created the splice, potentially making it a performance bottleneck.

Replicated Splice: To address these issues, 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.

Split Splice: 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 above, 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 allowing a backend server to directly send a response to a client without it having to pass through a proxy.

A TCP splice can be viewed as a combination of two unidirectional 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 half-splices is similar in spirit to the splitting of a TCP connection 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.

4 Flexible Server Configurations

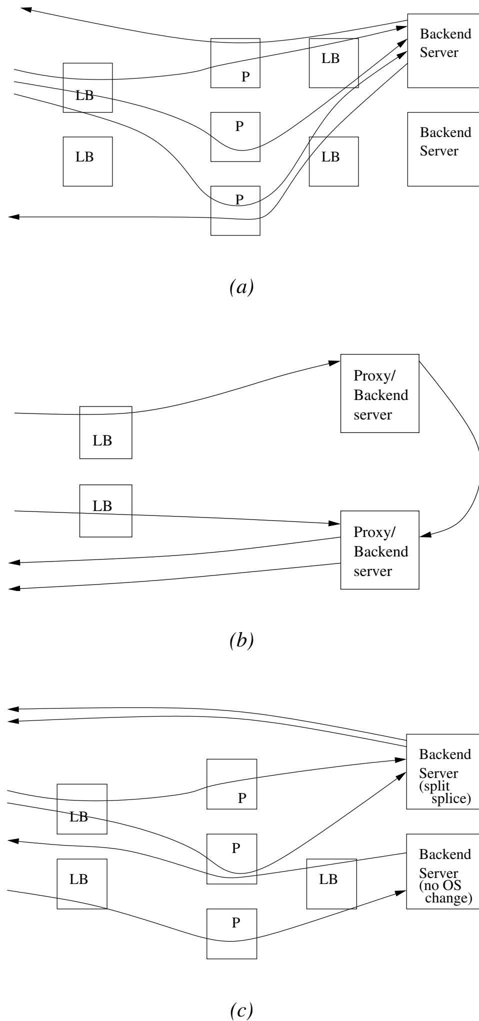

Figure 2: Some examples of possible server configurations. (a) Separate proxy machines; no backend server changes required. The figure shows the paths of packets belonging to a single connection. Packets in both directions need to pass through a proxy, but it could be any proxy. (b) Proxy and backend 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, one on each backend server.

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 backend server functionalities on the same set of machines, not changing the OS of backend servers at all, or even a mixed configuration where the OS of some backend servers is modified, while no changes are made to the OS of other backend servers. Figure 2 illustrates three server configurations.

In the first configuration, the proxy functionality is implemented on dedicated machines, and no changes are required to OS on backend 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 photos (similar to Flickr [12]). Replication of proxies (and hence the TCP splice) in this configuration results in high throughput, as the data from clients to servers can pass through separate proxies concurrently. Also, a proxy failure is transparently tolerated, since all subsequent data is forwarded via other proxies. Our experimental results, described in Section 8, confirm both of these features.

In the second configuration, the proxy functionality and backend server functionality are implemented on the same set of machines, potentially saving HW resources for the organization. OS changes are required on these machines. This configuration is suitable for organizations that do not require separate proxy machines, and, for web service applications that perform bulk data transfer to a client.

Finally, the third configuration is a mixed configuration. There are a small number of dedicated proxy machines, as required by an organization for security, and the OS of only some of the backend servers is changed. The OS of backend 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 backend servers that do not send bulk data to the clients remains unchanged.

Replication of TCP splice enables different requests or data from client to be routed via different proxies to a server. Failure of a proxy is tolerated by having subsequent packets routed via other proxies. However, response data from the servers to the clients still passes through one proxy for each connection. This obviously makes the proxy a failure bottleneck. This problem is addressed by using the split splice enhancement that allows a server to bypass proxies while sending a response. Split splice however requires modifications to a server. In situations where server modification is not feasible, a stateless LB, identical to the one used prior to the proxy stage, can be employed.

5 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 backend 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 which can exist as a separate box, or be

co-located with one or more routers or proxies. In either case, however, the packets are not modified in any way. For (2), a proxy acts as a L7 load-balancer.

5.1 L3 Load Balancer

There is an extensive body of research on L3/L4 load balancing. A survey of load balancing techniques, used in web server architectures, is provided in [8, 9]. 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 and maintains a mapping of the connection to that server. All subsequent packets belonging to that connection are sent to the same server.

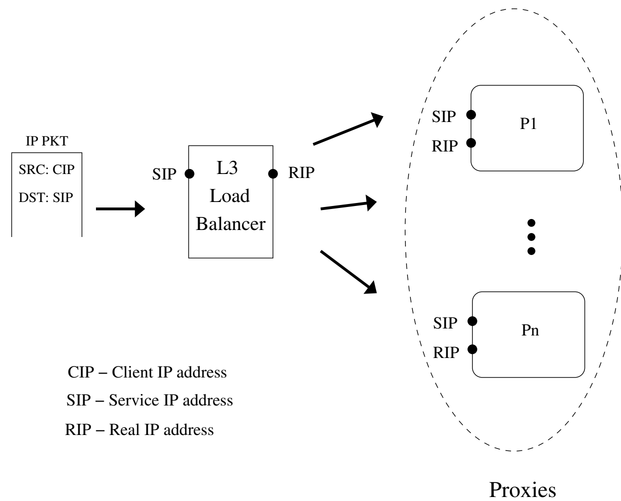

Figure 3: A layer 3 load balancer.

Our load balancer is shown in Figure 3. It is assigned a service IP address, which is the address that is advertised to the clients. All the proxies are also assigned the service IP address. 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 packets to it; (2) as part of the HB, a proxy sends a "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 the LB and all the proxies are in the same subnet. This requirement can be relaxed by the use of IP tunneling (although that may result 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 $r \in [\sum_1^{i-1} W_j', \sum_1^i W_j']$.

6 Proxy Architecture

Distributing and replicating TCP splice state information among all proxies allows packets belonging to a connection to flow through any of the proxies once the splice is established. 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 that accepted the initial TCP connection. Notice that a TCP splice is created only after receiving this first client request (which may be several TCP segments long).

To ensure that before a connection is spliced, all segments are received by the same proxy, and, to distribute the load of creating spliced connections evenly, the set of proxy machines are divided into 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 4 shows the sequence of steps involved in handling a client request. These steps are described below:

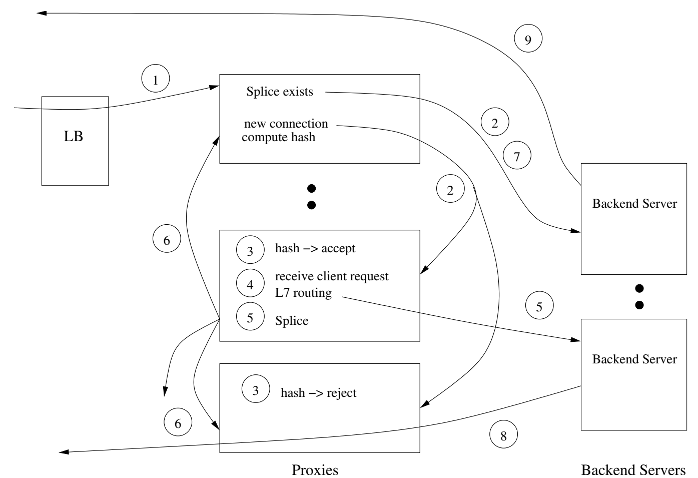

Figure 4: Sequence of steps involved in handling a client request.

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 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 received the complete 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 can now be spliced by any proxy to the backend server.

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.

7 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 5. 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.

7.1 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

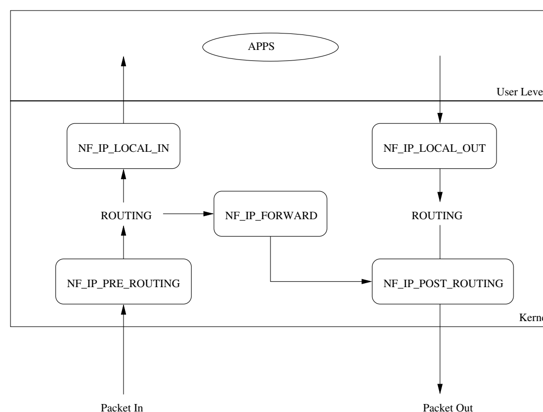

Figure 5: Netfilter allows functions to be registered at five different hooks in the network stack.

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.

For the experiments that follow, the load balancing algorithm used by the LB is round-robin. It sends 40 packets to a particular proxy before switching to the next one.

7.2 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 was implemented that — accepts a client connection, reads its request, performs L7-routing to connect to a server, passes splicing state information to other proxies, and then, finally, splices the client and server TCP connections. The significant parts of the proxy implementation are shown in Figure 6.

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`.

7.3 Backend Server

There is an optional modification that can be made to 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`.

The mechanism of sending the splice state information to a backend server and that of establishing a split splice there is

```
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 6: Pseudocode of the proxy implementation.

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.

8 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.

8.1 Experimental Setup

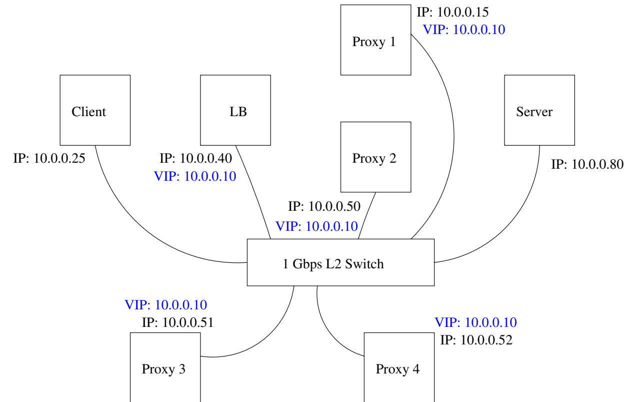

Figure 7: Experimental setup.

We use seven machines for our experiments. Specific details of the machines are listed in Table 1. The experimental setup is shown in Figure 7. The machines are connected together using an 8 port, 1 Gbps D-Link Ethernet Switch. Although we use a separate machine as a load balancer, it is not necessary to do so. In our experiments the LB was not a bottleneck, confirming the results in [5]. In larger systems, as mentioned in Section 5, the load balancing function could be integrated with the routers or the proxies.

The client sends requests to the virtual service IP address (10.0.0.10), which is assigned to the LB and all the proxies.

Client:	Shuttle xpc, Dual Pentium 4 – 3 GHz, 512 MB, 1 Gbps
LB:	Shuttle xpc, Dual Pentium 4 – 3 GHz, 512 MB, 1 Gbps
Proxy 1:	Dell Inspiron 9300, CPU 1.6 GHz, 512MB, 1 Gbps
Proxy 2:	HP DX2000, CPU 2.66 GHz, 128MB, 1 Gbps
Proxy 3:	HP DX2000, CPU 2.66 GHz, 128MB, 1 Gbps
Proxy 4:	HP DX2000, CPU 2.66 GHz, 128MB, 1 Gbps
Server:	Shuttle xpc, Dual Pentium 4 – 3 GHz, 512 MB, 1 Gbps

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

Since the load balancer is on the same LAN as the proxies in our setup, this can lead to a conflict when the client sends out an ARP for the VIP. 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 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.

8.2 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 two files sizes: 10 MB and 50 MB. The reason behind these choices is that these could represent the sizes of music, picture, or video files that a user may upload to a server through her web browser. Thus, in all, for 100 connections, 1 GB and 5 GB are transferred in the two cases, respectively.

For both of these file sizes, we conduct experiments for two different configurations: (1) a base configuration (C–S), where the client machine is directly connected to the server through the switch; and, (2) proxy configurations (C–LB–nP–S), where the client-server connection goes through a L3 load balancer (Section 5.1) and a proxy (Section 6). The base configuration is used as a baseline for comparing the performance of the proxy architectures. In the proxy configurations, the number of proxies is varied from 1 to 4.

Since we are interested in measuring the scalability of the proxies, we monitored the CPU and memory usage of the client, server, and the load balancer machines during each of the experiments to make sure that none of them become a bottleneck. Note that to prevent such a situation from occurring we have used machines with more powerful CPUs as the client, LB and server. This also makes the configuration more representative of a real-life situation, where there will be a multitude of clients and servers. `gnome-system-monitor` was run on these machines to observe the graph (updated every sec) of their CPU and memory usage.

Furthermore, 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.

8.2.1 Discussion of results

The results are summarized in Tables 2 and 3 for 100 concurrent PUTs of sizes 10 MB and 50 MB, respectively. The average, min and max times in these tables indicate the data transfer times (excluding connection setup time) for a single connection and are averaged over three runs. Since the data transfer for the 100 connections is started simultaneously, the total time taken is the same as the time taken by the slowest connection (max time). The corresponding network throughputs achieved are also listed. The last column in the tables shows the impact on the network throughput of increasing the number of proxies. When the number of proxies is increased from 1 to 2, ideally, if the system scales linearly by the same factor, the throughput should also double. In our case, the throughput increases by about 74% (from 125 to 217 Mbps, and 126 to 219 Mbps, in the two cases, respectively). Similarly, with three–proxy configuration, the percentage increase over two–proxies configuration should be 50%, but it is actually around 30% and 32%. Finally, for the four–proxy configuration, the increase is 18% and 23%. Thus, the increase in network throughput with the increase in the number of proxies, shown in Figure 8, is almost linear up to at least four proxies. As the number of proxies is further increased, the throughput is likely to hit a knee and flatten out. In the future, we would like to add more proxies and determine how close the network throughput of the proxy architecture can get to that of the base configuration (client and server connected directly).

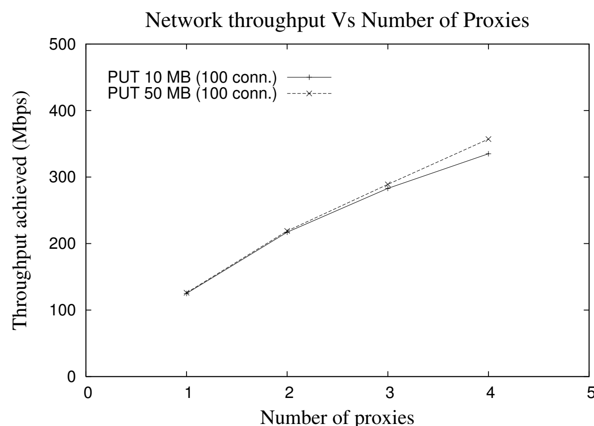

Figure 8: The plot shows the increase in the network throughput as more proxies are added.

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.

8.3 Proxy Failovers

The objective of these experiments is to show that the impact of a proxy failure on the users is negligible. In these ex-

Config.	Time Taken (sec.)			Throu-ghput (Mbps)	Throughput increase over previous proxy config.
	Ave.	Min	Max		
C–S	8.14	5.38	8.93	895	n/a
C–LB–P–S	59.56	43.74	64.17	125	n/a
C–LB–2P–S	34.27	24.55	36.90	217	73.6%
C–LB–3P–S	24.74	18.22	28.25	283	30.4%
C–LB–4P–S	20.47	14.91	23.88	335	18.3%

Table 2: Experimental results for 100 concurrent client PUT requests of a file of size 10 MB

Config.	Time Taken (sec.)			Throu-ghput (Mbps)	Throughput increase over previous proxy config.
	Ave.	Min	Max		
C–S	43.10	38.31	44.87	892	n/a
C–LB–P–S	307.25	266.25	318.73	126	n/a
C–LB–2P–S	177.12	160.49	182.94	219	74.2%
C–LB–3P–S	129.19	107.86	138.25	289	32.3%
C–LB–4P–S	106.87	92.83	112.07	357	23.4%

Table 3: Experimental results for 100 concurrent client PUT requests of a file of size 50 MB

periments, the client sends 100 simultaneous PUT requests to the server, similar to the experiments in the previous section. Note that these experiments are done with a 100 Mbps switch, and, hence, the network throughputs are lower; however, this does not impact the results of these experiments.

Each request transfers 10 MB to the server. Experiments are performed with a two proxy configuration (C–LB–2P–S). 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 removes 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, the non-failure case takes about 90 sec., so we failed a proxy at about 45 sec. after the start.

8.3.1 Discussion of results

The results are summarized in Table 4. 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.

HB Interval	Time (in sec.)					Network Throughput (Mbps)
	Avg.	Min	Max	Std. dev	Total	
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 4: Experimental results for 100 concurrent PUTs of 10 MB with a proxy failure during the transfer.

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 may cause the client to invoke TCP congestion control. We also calculated the network throughput achieved in each of these cases. The network throughput, of course, decreases when there is a proxy failure. This is because of packet loss during failure detection, fewer number of usable proxies for the remaining data transfer, and, if the failure has caused enough packet loss, TCP's recovery from congestion control. However, we do observe that the throughput remains relatively large when the HB interval is short. The best results obtained are for HB interval of 200ms, where the failure is detected between 400 and 600ms. The average network throughput in this failure case is only 5.3% less than the no-failure case.

8.4 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, 100 concurrent connections are created by the client, and, the HTTP GET method is used to download files from the server. Similar to the earlier experiments, files of two different sizes are transferred: 10 MB and 50 MB.

Three different configurations are used. (1) C–S, where the client directly connects to the server; (2) C–LB–P–S, where one proxy is used; other than the reversal in the direction of the data, this is identical to the PUT experiment performed with this configuration in Section 8.2; 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.

8.4.1 Discussion of results

Tables 5 and 6 show the performance measured for 100 concurrent GETs of sizes 10 MB and 50 MB, respectively. The results show that with split-splice (C–LB–P–S*), we get throughputs that are almost indistinguishable from those of the base case (C–S). This is to be expected since the data is directly sent to the client from the server. Furthermore, split splice is most efficient when bulk data has to be transferred from a server to a client.

Config.	Time Taken (sec.)			Throughput (Mbps)
	Average	Min	Max	
C–S	8.14	5.38	8.93	895
C–LB–P–S	59.56	43.74	64.17	125
C–LB–P–S*	8.69	4.51	9.23	866

Table 5: Experimental results for 100 concurrent GET of 10 MB with *split splice* at the server.

Config.	Time Taken (sec.)			Throughput
	Average	Min	Max	(Mbps)
C–S	43.10	38.31	44.87	892
C–LB–P–S	307.25	266.25	318.73	126
C–LB–P–S*	43.26	35.34	44.99	889

Table 6: Experimental results for 100 concurrent GET of 50 MB with *split splice* at the server.

Also, we observed that the CPU utilization of the server increases substantially when it is performing split splice. If a server does not have adequate CPU resources, the transfer rate can get CPU bound. However, we did not encounter this. While performing split splice, our server was running at about 70% CPU utilization (one CPU was at about 100% and the other at about 40%).

9 Conclusions and Future Work

User applications that usually run on a machine locally are beginning to be offered remotely through a web browser. Furthermore, users are increasingly using servers to store large amounts of data, such as images and videos. These 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 two areas of future work that we are pursuing. The first area 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 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 is a more sophisticated load balancing algorithm in the load balancer (LB). In the current prototype, the LB uses round-robin to distribute the load among the proxies, rather than considering the workload on the proxies as described in Section 5.1.

The second area of future work 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 tolerate a server failure such that it is completely seamless to the user. With the newly emerging applications, providing client-transparent, seamless service to the clients in the face of backend 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.