Half–pipe Anchoring: An Efficient Technique for Multiple Connection Handoff[*]

Ravi Kokku[‡]   Ramakrishnan Rajamony[†]   Lorenzo Alvisi[‡]   Harrick Vin[‡]

[‡] Dept. of Computer Sciences
University of Texas, Austin
{rkoku,lorenzo,vin}@cs.utexas.edu

[†]Austin Research Lab
IBM, Austin
rajamony@us.ibm.com

Abstract

We present half-pipe anchoring, *a technique that enables multiple connection handoff mechanisms that are efficient and easy to implement. In a server cluster, these mechanisms result in better resource utilization and improved scalability. More importantly, half-pipe anchoring supports the only connection handoff mechanism that operates efficiently in heterogeneous clusters composed of specialized nodes. The key idea behind our approach is to decouple the two unidirectional half-pipes that make up a TCP connection between a client and a cluster. We anchor the unidirectional half-pipe from the client to the cluster at a* designated server *while allowing the half-pipe from the cluster to the client to migrate on a per-request basis to an* optimal server *where the request is best serviced. We describe the design and implementation of a multiple connection handoff mechanism in the Linux kernel that demonstrates the benefits of our technique.*

1. Introduction

Three trends characterize today's content servers that host services such as e-mail, e-commerce, and search engines. First, services are increasingly providing their clients with personalized content. Recent studies [18, 19] reveal that the percentage of companies adopting secure-content technologies—which typically generate dynamic or personalized content—was 76% in 2001, while those using personalized content from XML-based applications was 67%.

Second, partly in response to the added complexity and diversity of clients' requests brought by personalization, cluster architectures are changing. No more a collection of identical nodes, clusters are increasingly structured around specialized nodes, customized to efficiently perform specific subsets of the tasks involved in processing a request [21, 34]. Node specialization improves efficiency and scalability of the cluster and reduces overall energy consumption [3, 11, 15, 16]. For example, nodes that predominantly serve static data can benefit from large amounts of memory, while a powerful processor is more beneficial for nodes serving compute-intensive dynamic content. Similarly, static requests can be serviced by an efficient in-kernel implementation of a server application like TUX [4], while dynamic requests that typically require complex cookie parsing or database accesses are better handled by a user-level server application like Apache [1]. Nodes can also be specialized to perform specific parts of request processing efficiently [29, 30] where some nodes can handle network level packet processing, some can handle data serving and caching and some can handle complex database querying. Even at the application level, complex applications like MultECommerce [28] specialize functionalities performed by nodes to achieve scalable systems.

Third, clients can send multiple, and potentially very different requests to the same server over a single persistent TCP connection [9, 25]. Persistent connections eliminate the overhead involved in setting up and tearing down a connection for each request. Unfortunately, they also make it hard to direct each request coming on the connection to the cluster node that is best suited to service it.

In this paper, we present *half-pipe anchoring*, a novel technique that allows individual requests coming on a persistent connection to be processed at the cluster node that is best equipped to serve them. Half-pipe anchoring is based on the observation that a TCP connection can be viewed as two half-pipes, one from the cluster to the client (data pipe) and one from the client to the cluster (control pipe). Our approach anchors the control pipe at one server, which we call the designated server, while allowing the data pipe to migrate on a per-request basis to the server where the request is best serviced. We obtain the coordination needed to allow the control pipe and the data pipe to reside on different

nodes through a simple protocol, which we call *Split-stack*.

Half-pipe anchoring goes beyond existing connection handoff protocols [5, 6, 20, 26, 27, 31, 32, 33] in at least two respects. First, it supports efficient *multiple* handoffs of the same connection by allowing request processing to be pipelined. Thus, a node processing request r_1 can hand off the connection to a new node for processing the next request r_2 before all TCP traffic related to r_1 has been received by the client. Second, and more importantly, half-pipe anchoring is the only connection handoff mechanism designed to operate efficiently in heterogeneous clusters composed of specialized nodes. For example, half-pipe anchoring enables a server architecture where nodes optimized for request processing do not perform any data serving and vice versa.

As a proof of concept, we have built a prototype implementation of half-pipe anchoring in the Linux kernel. Our experiments show that the prototype supports multiple handoffs with minimum overhead. For example, for a response size of 15 KB, the prototype incurs an overhead of 16% in the response time perceived by a client in a LAN. The overhead drops to 0.36% in a WAN environment with an average round-trip latency of 40 ms. We compare the performance of our prototype with an existing multiple handoff solution (KNITS) [27] and find that the overhead incurred by our solution is atmost one-fourth that of KNITS.

The rest of the paper is organized as follows. Section 2 explores the design space of existing solutions and then presents a case for the requirement of a novel multiple connection handoff mechanism for today's content servers. Section 3 presents half-pipe anchoring, our approach to building such a mechanism. Section 4 provides a description of our prototype implementation. Section 5 presents an evaluation of our prototype and Section 6 concludes.

2. Design Space

The problem of directing a request to the server best suited to process it can be addressed using *end-to-end* or *cluster-based* approaches. End-to-end approaches use content-authoring and/or transport-level mechanisms to process requests at the "right" server. Two kinds of end-to-end approaches have been proposed in the literature.

The first approach consists of authoring content and services so that clients can send each request directly to the appropriate server. This approach is simple but has multiple drawbacks. First, it requires a client to open multiple connections with servers within a server cluster, wasting both server and network resources [14, 25]. Second, it exposes the configuration details of the server cluster to the content authoring process (and hence to the clients). Finally, it prevents the cluster from achieving fine-grained load balancing across servers.
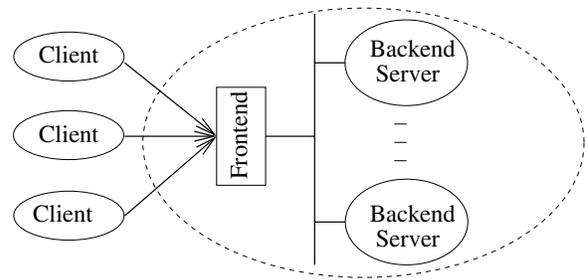

Figure 1. A simplified cluster architecture

The second approach uses a transport-level mechanism [31, 32] to migrate a client connection from server to server on a per-request basis. A nice feature of this approach is that it does not require any change in the client's application. However, it has other serious drawbacks. First, it requires clients to support connection migration at the transport level, which makes it hard to deploy. Second, because large WAN delays result in relatively high migration latencies, the approach is impractical for short web transfers where the time necessary to migrate the connection is higher than the time required to transfer the data.

The drawbacks of end-to-end approaches make it advantageous to consider *cluster-based* mechanisms. To simplify the exposition of these mechanisms, consider a server cluster architecture with a single frontend node and a collection of backend servers (Figure 1). The frontend node provides a single-IP-address view of the entire server cluster to the clients and performs some additional functions such as load balancing and content-aware request distribution [26]. The backend nodes service client requests. The frontend and the backend servers are connected using an internal high-speed network. We refer to the backend server that holds the connection from a client as the *designated* server; the backend server that is best-suited to service the client request we call instead the *optimal* server.

To classify and explore systematically the design space for cluster-based solutions, we partition the task of servicing requests by optimal servers into three components.

1. Layer-7 switching: This component processes client requests and determines the optimal server for servicing the request based on criteria such as server load, availability of content or service at the server, etc.

2. Connection management: This component manages the interactions between the client, the designated server, and the optimal server to ensure that the response is correctly received by the client.

3. Cluster transparency: This component is responsible for sending responses from the optimal server to the client. In particular, this component modifies packets transmitted from the optimal server such that, from the client's perspective, responses appear to come from the frontend
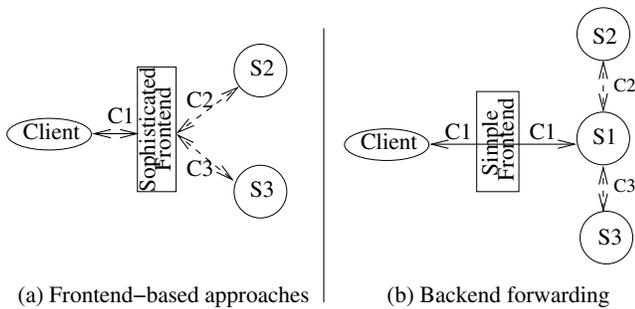
(a) Frontend–based approaches   (b) Backend forwarding

Figure 2. Cluster based mechanisms

node. By modifying packets, handing off the responsibility to service a request from the designated server to the optimal server is completely transparent to the client.

We now classify and compare different cluster-based mechanisms based on where in the cluster (frontend or backend) these three components reside. We evaluate the relative merits of these approaches in terms of (1) the scalability of the server cluster platform, (2) the efficiency of cluster resource utilization, and (3) other constraints that the approaches may impose on the cluster architecture.

Frontend-based Approaches In the frontend-based approach (Figure 2(a)), the frontend node accepts all client connections (C1), performs layer-7 switching for each request, sends request to and receives response from the optimal server on persistent connections (C2 and C3), and finally forwards the response to the client. Relaying frontend architectures such as the commercially available Redline web accelerators [5] fall into this category.

The main advantage of this approach is that it encapsulates in the frontend all the components needed for servicing requests at their optimal servers. The backend servers are completely unaware of the overall operation of the server cluster (and hence, server clusters can be put together from off-the-shelf components). The main disadvantage of this approach is that the frontend node becomes a bottleneck and limits the overall scalability of the server cluster. Although techniques such as TCP splicing [17] reduce the overhead of relaying data through the frontend node, the complexity of layer-7 switching and connection management make the frontend the limiting factor in cluster scalability [13].

Hybrid Approaches These solutions split the three components between the frontend node and the backend servers. KNITS [27] is one such solution. In KNITS, layer-7 switching is done at the backend servers, while the frontend node is responsible for connection management and cluster transparency. An incoming connection is sprayed by the frontend onto a designated backend server chosen by a simple distribution policy to balance load on the backend servers. After identifying the optimal server for a request on the connection, the designated server forwards the request to the

optimal server and informs the frontend of this handoff. The frontend then splices together the connection from the optimal server to the frontend with the connection from the designated server to the client, so as to keep the handoff transparent to the client. This approach is easy to deploy since it requires no modifications to the kernels of the backend servers. However, since the connection management and cluster transparency functions continue to reside at the frontend, the frontend limits the scalability of the cluster. We quantify the frontend overhead in KNITS in Section 5.

Backend-based Approaches In this model, all three components are performed at the backend servers. The frontend node merely performs the layer-4 functions by spraying connections to designated backend servers in a round-robin manner. It has been shown that a frontend switch that performs only the layer-4 functions is significantly more scalable than a frontend that performs extra processing such as content-aware request distribution and TCP-splicing [13].

The simplest instantiation of the backend approach is called *backend forwarding* (Figure 2(b)). In this case, once a backend server receives a request, it (1) identifies an optimal server to service the request, (2) forwards the request to the optimal server over a persistent connection (C2 or C3), (3) receives the response from the optimal server, and (4) forwards the response to the client over the client-cluster connection (C1). Backend forwarding is simple to implement, and is more scalable than frontend-based approaches. However, this approach requires data to be forwarded to the client through the designated server. This wastes processor [22], memory and network bandwidth at the cluster, which also leads to increased power consumption.

The overhead inherent in the backend forwarding model can be eliminated if the connection from the client to the designated server is handed off to the optimal server. In this model, the optimal server directly transmits the responses to the client. However, the optimal server has to now perform the layer-7 switching for any subsequent requests on that connection. Migrating layer-7 switching to the optimal server has a significant limitation. It requires each backend server to be capable of processing all incoming requests and hence can not be used in server clusters created from heterogeneous, specialized components.

In summary, connection handoff, a backend-based approach, is likely to be the most scalable among all the alternative architectures discussed above. However, to be viable, the connection handoff architecture (1) should be applicable to clusters created from heterogeneous, specialized components, and (2) should efficiently support multiple connection handoffs per connection to efficiently handle multiple requests with widely different requirements over the same connection. In what follows, we propose a novel architecture that meets both of these requirements.
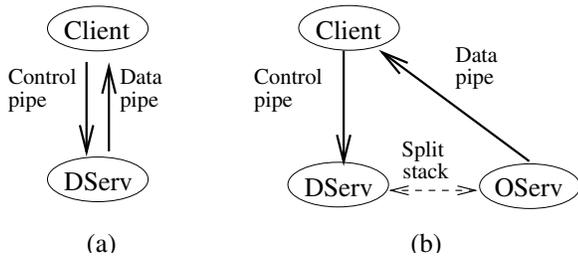
3. Half-pipe Anchoring

Figure 3. Separating the two half-pipes

A TCP connection can be viewed as a combination of two unidirectional half-pipes—a *control pipe* and a *data pipe*. The control pipe carries client requests and acknowledgments to the server. The data pipe carries responses from the server to the client. At either end of the connection, TCP achieves flow control and reliability on one half-pipe based on protocol messages (acknowledgments) that it receives from the other half-pipe. Traditionally, the two half-pipes of a TCP connection reside on the same node at either ends (Figure 3(a)).

The requirement that the two half-pipes be co-located can be relaxed as long as flow control and reliability across the half-pipes are maintained. Figure 3(b) shows the two half-pipes separated. Here, the control pipe is anchored at the designated server (DServ) and a data pipe is instantiated at the optimal server (OServ). OServ sends back a response to the client over the data pipe while DServ receives and processes new requests (for which it determines new optimal servers) and acknowledgments over the control pipe. The coordination between the control pipe and the data pipe (for flow control and reliability) is exchanged using *Split-stack*, a lightweight communication protocol that we describe in Section 4.

Support for Heterogeneity Anchoring the control pipe at one server node for the connection duration centralizes layer-7 processing at DServ. By relaxing the requirement that every server be capable of layer-7 processing, half-pipe anchoring simplifies the design and deployment of heterogeneous server clusters. For instance, half-pipe anchoring makes it possible to build a web serving cluster in which different specialized nodes are responsible for parsing client requests, serving static data and returning dynamic content.

Multiple Connection Handoffs Consider two successive requests, r_1 and r_2, received by DServ on the same persistent connection from a client. Two scenarios can occur: (1) r_2 reaches DServ after the client has completely acknowledged the response to r_1, or (2) r_2 reaches DServ while r_1

is still being served.

Half-pipe anchoring trivially handles the first scenario. Once r_1 is serviced completely, DServ has a consistent state of the connection. When it receives r_2, DServ simply instantiates a new data pipe at an appropriate optimal server.

A simple way to handle the second scenario would be to prevent DServ from handing off the connection until all the operations related to previous requests (data transmission and ack receiving) are completed. However, this approach wastes network bandwidth by deliberately draining the data pipe before handing off the connection. In fact, Aron et. al. have identified pipe-draining as a potential problem that a connection handoff protocol must address [12].

Half-pipe anchoring prevents data pipe drains during connection handoff. As soon as the optimal server for r_1 finishes sending the last data packet, it informs DServ that it is done. At this point, while the data packets from r_1's optimal server are still unacknowledged, DServ can direct a second optimal server to service r_2, thereby preventing the data pipe from draining. Anchoring the half-pipe from the client enables DServ to forward the client's acknowledgments to the appropriate optimal server.

4. Prototype: The Split-stack
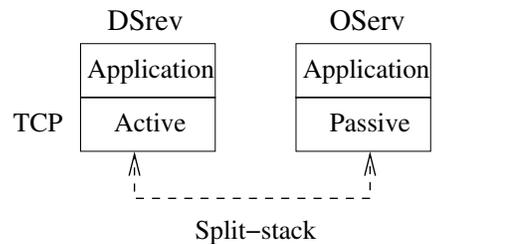
(a) Active and passive layers in Split-stack
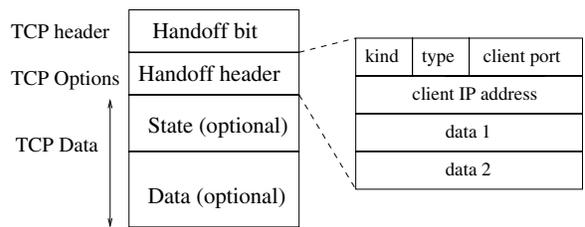
(b) Message structure

Figure 4. Split-stack protocol details

Split-stack is a light-weight communication protocol that coordinates the control and data pipes when they are separated on different nodes. In order to better explain the Split-stack protocol, we use the following terminology: the TCP layer on the designated server is called the *active layer* and that on the optimal server is called the *passive layer*. (Figure 4(a)). The active layer holds the control pipe, performs

request and ack demultiplexing and controls the data pipe instances created on the optimal servers. The passive layer receives commands from the active layer to create, monitor and destroy instances of data pipes. We have implemented a prototype Split-stack in the TCP stack of the Linux 2.4.10 kernel.

4.1. Message Structure

To exchange handoff messages between the active and passive layers, we extended the TCP header and data as shown in figure 4(b). The active and passive layers identify handoff messages using a *handoff bit* (one of the reserved bits in the TCP header [10]). All packets that enter a node's TCP layer are intercepted, and checked for the handoff bit. If this bit is set, the handoff message is handled appropriately according to the message type; otherwise the packets traverse the vanilla (unmodified) TCP stack.

The handoff header is sent as a TCP option and contains six fields. The *kind* field identifies that the option is for handoff messages. The *type* field defines the type of the message. It could be one of SS_SETUP, SS_DONE, SS_CTRL, SS_RESET or SS_FAILURE. *Client port* and *client IP address* identify the client whose connection is being handed-off. *Data 1* and *Data 2* are fields used for carrying data specific to the message types. The state and data parts of TCP data represent the connection state and request data carried in SS_SETUP respectively.

To keep track of the data pipe instances created by handoffs for sending subsequent messages, the active layer maintains a list of the following five–tuple entries corresponding to each data pipe:

<OServ.IP, client.IP, client.port, Seq.start, Seq.end>

The handoff messages are sent as IP packets. We currently handle loss of handoff packets in the internal network using timeouts. On a timeout, the handoff just fails. Making the handoff reliable is straightforward with some additional functionality. However, since packet losses are rare in a tightly-coupled LAN environment, we tradeoff the rarely occurring handoff failures for a simpler design.

4.2. Sequence of Actions

Figure 5 depicts the message sequence chart of handling a client request remotely (some of the irrelevant messages like 3-way handshake between client and designated server, are not included in the chart). The following steps are executed in order to process a request remotely:

1. Layer-7 switching: On the arrival of a client request, the application on the designated server (DServ Appl) determines the optimal server where the request should ideally be handled. The DServ Appl then calls the handoff system call to handoff the request to the optimal server.

2. Connection management: The active layer sends a SS_SETUP message to the passive layer in response to the execution of a handoff system call by the DServ Appl. The current TCP connection state of the socket is sent on the message to the passive layer on the optimal server, along with the request data provided by the application in the handoff call. In Linux, all TCP connection state is stored in the *tcp_opt* structure. The active layer also stores a five-tuple entry corresponding to the passive layer for sending control and reset messages.

On the optimal server, the passive layer receives the SS_SETUP message and creates a new socket using the provided connection state. The socket is created in the established state *without* a TCP three-way handshake. The passive layer then provides the request data to the OServ Appl.

3. Cluster transparency: When the OServ Appl sends data to the passive layer to be sent to the client, the passive layer sends the response to the client after modifying the sender address in the TCP header to indicate that the data actually originated from the designated server. The OServ Appl closes the connection after sending the response to the passive layer. The passive layer, on receiving the close, does not trigger the normal TCP-FIN exchange [10] with the client and instead returns success to the OServ Appl.

4. SS_DONE and SS_FAILURE: Once all the data is sent out to the client, the passive layer sends an SS_DONE message to the active layer with the sequence number of the last byte sent. This is when the *Seq.end* field gets updated in the entry on the active layer corresponding to the data pipe. If a setup failure occurs because no OServ Appl exists or because no memory is available on the optimal server, then an SS_FAILURE is sent instead. The active layer returns from the handoff system call indicating success or failure to the application based on the message it received.

5. Ack and request demultiplexing: Since the control pipe is anchored at the designated server, all acks from the client arrive at the active layer. For every ack that the active layer receives, based on the data pipe entries that it has, it generates a SS_CTRL message to the respective passive layer. When this message is received, the passive layer converts the message into an ack and injects it up the vanilla TCP stack. The TCP stack on the optimal server views the ack as if it were sent from the client. Based on the ack received, new packets are sent out to the client, following the normal TCP actions on the optimal server. If the SS_DONE message has not yet been received from a particular passive layer (i.e. *Seq.end* is not yet known), but the active received an ack from the client that acknowledges data that is potentially sent out by this passive, then the active still sends a control message to the passive.

Each SS_CTRL message also contains the correct sequence number of the most recent request data received from the client, so that subsequent response data packets
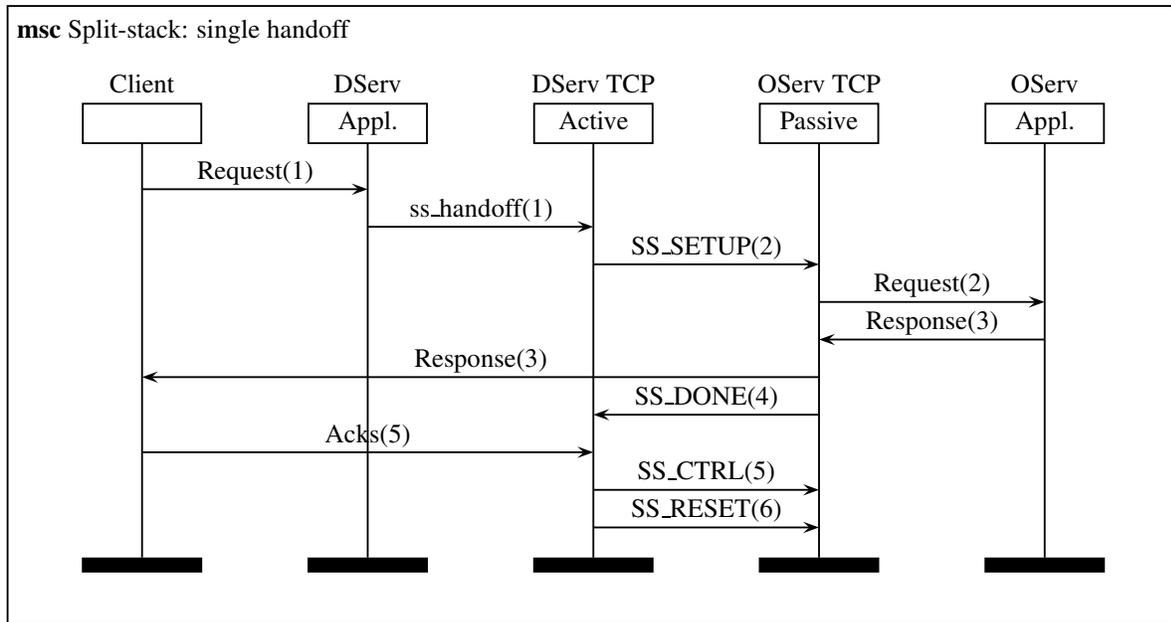
Figure 5. Message sequence chart for single connection handoff using Split-stack

from an optimal server send correct acks to the client.

As a part of the half-pipe anchoring technique, subsequent requests that have already been received are not sent to the new optimal server. Instead, they are processed locally at the designated server. This choice minimizes the overhead of the handoff.

6. SS_RESET: If an ack acknowledges all the packets send by a passive, an SS_RESET is sent to the passive layer. The passive layer destroys the socket in response.

4.3. Concurrent Handoffs

Consider the case where a new request arrives while a handoff for another request is already in progress. In order to minimize the request processing latency, the request should ideally get processed as soon as the DServ Appl processes the request. Split-Stack initiates the handoff for the next request as soon as SS_DONE is received for the previous request. This is the earliest time when an active layer can initiate a second handoff (only after receiving the SS_DONE message from the first passive layer does the active layer have complete information about the state of the connection). Pipelining request processing in this manner leads to efficient use of resources. However, one of the crucial challenges with handling concurrent handoffs is the correct handling of incoming acks and request data.

We handle the challenge based on the following intuition. A TCP acknowledgement serves two purposes. First, it serves as a *credit* to send more packets on the connection. Second, it informs the sender that a previously sent packet (which was saved to handle retransmissions) has been re-

ceived at the client, enabling the space occupied by the packet to be *reclaimed*. While credits to send more packets are meaningful only when there is more data to send back to the client, space reclamation must always be performed.

Split-Stack correctly handles request data and acknowledgements by explicitly separating these two functions. After transmitting all the response data to a client (but potentially before all data has been acknowledged), the optimal server for that request informs the active layer of the amount of data sent. Using this information and the acknowledgement sequence number, the passive layer that must reclaim space is identified. The active layer then sends an SS_CTRL message to this layer to reclaim packet(s). The Split-stack protocol permits only one optimal server to transmit new data packets to the client at a given point in time. Consequently, the active layer can easily identify which server should send a new data packet. This is also conveyed using an SS_CTRL message. When packet reclamation and data sending have to be done on the same server, the active layer sends only one SS_CTRL message. Finally, if all packets sent by an optimal server are acknowledged, the active layer sends an SS_RESET message to the passive layer.

4.4. API

When an application on the designated server determines that a request is better handled on a different node, it invokes the following system call:

ss_handoff(request, OServ-IP, OServ-port)

The handoff system call returns success or failure depending on whether the request was handled successfully

by the optimal server. On failure, the DServ Appl can either (1) handle the failed request locally, (2) handle the failed request at another remote optimal server node or (3) send an error response to the client. The DServ Appl can then proceed to receive and process the next request from the client.

The OServ Appl is unaware of the handoff and simply sends the response back on the connection. However, our implementation currently expects that the OServ Appl closes the connection when the request is served. This causes the passive layer to send the SS_DONE message.

4.5. Discussion

Our prototype implementation currently does not handle some options, such as TCP timestamps [7] and SACKs [8]. Although these TCP options are important in practice, they were not required to demonstrate the efficacy of half-pipe anchoring. However, since all packets from the client are received at the designated server and as the active layer completely controls the passive layer, any information related to TCP options can be easily communicated to the passive layers using SS_CTRL messages.

For example, to handle timestamps correctly, the designated server should ensure that the timestamp values used in the packets being sent to the client are monotonically increasing [7]. To achieve this, the designated server can piggyback its timestamp on SS_CTRL credits to send new packets. The optimal server can use this timestamp in the response packets. Similarly, SACK options can be interpreted and sent as retransmission requests in the SS_CTRL message. Our prototype correctly handles options exchanged during SYN exchange between the client and the designated server, such as the "Window scale" and MSS options. These options get reflected in the connection state that is sent to an optimal server in the SS_SETUP message.

An added advantage of building a connection handoff mechanism based on half-pipe anchoring is that it enables some nodes in the cluster to run a reduced and optimized communication stack (like the one explored by Abegnoli et al. to accelerate web servers [23]). Such a stack will only need to implement the functions of a passive layer, such as sending more packets, retransmitting packets, and reclaiming resources on the directions of the active layer.

With a minor extension to the SETUP_DONE message by allowing it to carry back data, our prototype can support dependent requests. Two requests are dependent if the state modified by one request is used by the other. However, the DServ and OServ applications will be responsible for transferring the required state back and forth using the underlying Split-stack protocol. The same holds true for secure http connections where the DServ security layer is responsible for transfering enough state to the OServ security layer to handle secure connections. More broadly, the Split-stack protocol is a transport level infrastructure that just supports remote request processing. Higher level protocols should explicitly manage the required state to operate correctly.

5. Prototype Performance

We present an evaluation of our prototype in this section. Our experiments were designed with three goals in mind: (1) to analyze the various overheads our prototype incurs in handling a request remotely in comparison to handling the request locally, (2) to show the efficacy of half-pipe anchoring in handling multiple handoffs and (3) to compare the performance of half-pipe anchoring with KNITS, an existing system that supports multiple handoffs.

5.1. Setup

Our experimental environment consists of 450 MHz Pentium II PCs used as servers and 933 MHz Pentium III PCs used as client and delay router, all interconnected using a switched 100 Mbps Ethernet. We chose the client to be more powerful than the servers to ensure that the client does not become a bottleneck during our throughput measurements. All servers run the Linux 2.4.10 kernel with our modifications to the TCP stack. We used Apache 1.3.19 as our server application. We made a minor modification to Apache to invoke the ss_handoff system call to handle a request remotely. On the client side, we used the Httperf workload generator [24] to drive our experiments. We modified Httperf to maintain a specified number of outstanding connections to the server.

5.2. Split-stack Overhead

By anchoring the half-pipe from the client at one node and sending response data from another node, the Split-stack protocol incurs extra overheads over the case where requests are processed on the node at which they arrive. We measure the extent of this overhead using two experiments.

The first experiment measures the overhead due to addition of the active and passive layers to the networking stack using the experimental setup in Figure 6(a). In the base case, a client opens a connection to a vanilla 2.4.10 system and sends one request. The connection is then closed by the client and only then, a next connection is opened. We measure the average connection lifetime over 3000 connections as the response size is varied. The line marked "direct" in Figure 7(a) shows the results of this experiment. We then replace the server with our modifications to the network stack and again measure the average connection lifetimes. This data is presented as the line marked "Direct with kernel mods" in Figure 7(a). Over the range of file sizes that we tested, the two cases are within 1% of each other, borne
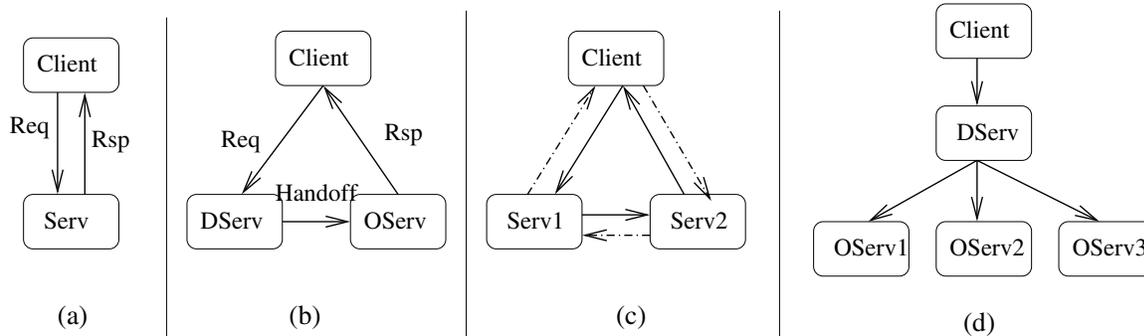
Figure 6. Experimental setup
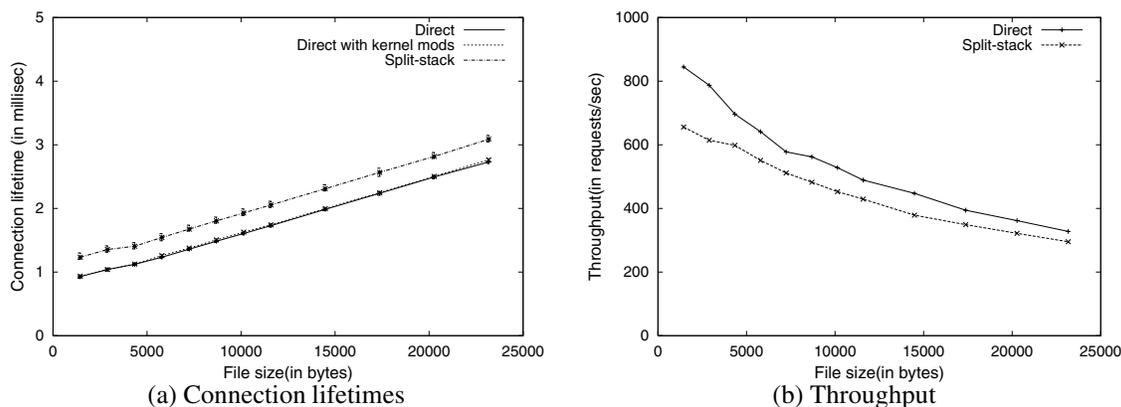

(a) Connection lifetimes (b) Throughput

Figure 7. Effect of varying response size

by the fact that the two lines lie almost on top of each other. This indicates that the modifications added to the base kernel (in the form of the active and passive layers) adds very little overhead to normal packet flow in the network stack.

The second experiment measures the overhead of the Split-stack protocol using the experimental setup in Figure 6(b). As in the previous case, we measure the average connection lifetimes for single–request connections as the response size is varied. In addition, we also measure the throughput perceived by the client. Figures 7(a) and 7(b) illustrate these results. In both figures, the "direct" line represents the case where a response is sent back from the server to which the request was sent. The "Split-stack" line represents the case where the client sends the request to the designated server, which then hands-off the connection to the optimal server that services the request. The gap between these two lines represents the overhead due to the Split-stack protocol. Most of the overhead is incurred in the initial setup, as indicated by the near-constant spacing between the two lines in Figure 7(a). For measuring the throughput, as depicted in Figure 7(b), we maintained 100 outstanding connections at the servers and let the tests (for each point) run for 1 minute. The effect on throughput is more for small file sizes which is again explained by the

Table 1. WAN delay *vs.* Connection lifetimes

WAN delay (in ms)	Direct (in ms)	Split-stack (in ms)	Overhead
LAN (0.2)	1.97	2.29	16.24%
2	12.23	12.73	4.08%
10	44.44	44.94	1.12%
20	84.47	85.08	0.72%
40	164.54	165.14	0.36%

constant nature of the Split-stack overhead.

Furthermore, the Split-stack handoff overhead becomes negligible when we consider a Wide Area Network (WAN) environment where packet delays are often a couple of orders of magnitude more than in a LAN environment. We used the NISTNet delay router [2] between the client and the severs to introduce delays on both request and response paths. Table 1 shows the connection lifetimes perceived by a client requesting a 15KB document when the WAN delay is varied. The near-constant difference in connection lifetimes (irrespective of the WAN delay) demonstrates that most of the overhead is incurred in the handoff initiation.

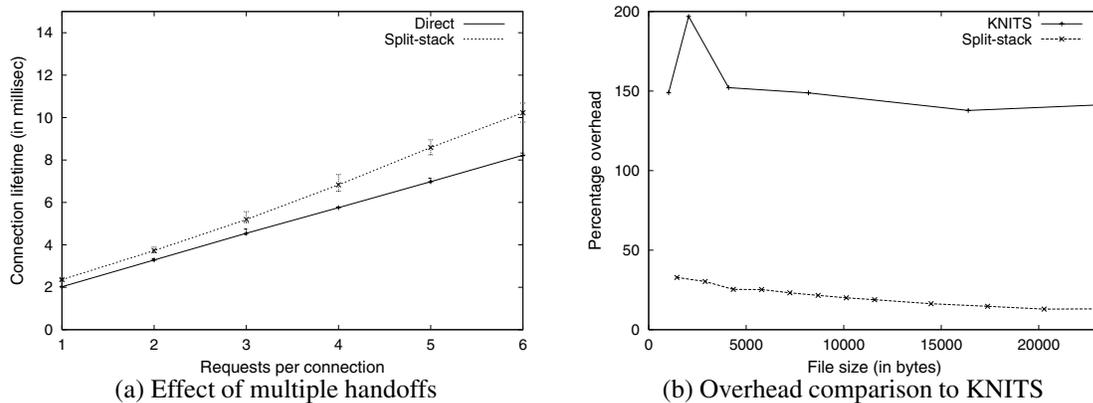In general, handoff mechanisms use more resources than

(a) Effect of multiple handoffs   (b) Overhead comparison to KNITS

Figure 8.

the "direct" case due to the costs of initiating the handoff, forwarding acknowledgements, and duplicated application-level processing. We provide an upper bound on these costs for the Split-stack prototype using the experimental setup in Figure 6(c). As before, the client sends requests to two servers. In the "direct" case, requests are processed where they arrive. The Split-stack case assumes a worst case scenario where a request is never serviced locally. In both cases, we measure the throughput (requests processed per unit time) of the two-server cluster.

The results of this experiment show that while the direct case is able to achieve a maximum throughput of 1348 reqs/sec, the Split-stack cluster can process only 796 reqs/sec, corresponding to a 40% degradation in throughput. Most of this overhead can be attributed to the duplicated application processing in our setup. Our prototype uses the Apache webserver on both the designated and optimal servers. Using a customized server application on the designated server that is optimized for request processing could reduce some of this overhead. Some of the overhead can also be attributed to our favoring code portability across kernel versions over raw performance. Another conclusion we can draw from the observations in this experiment is that since the overhead involved in application level processing and the handoff mechanism could be significant, they are better performed on the backends.

5.3. Multiple Concurrent Handoffs

We crafted a different experiment to measure Split-stack's ability to concurrently perform multiple handoffs on a connection. The setup used for this experiment is shown in Figure 6(d). A single DServ receives all requests from the client and distributes them among OServ1, OServ2 and OServ3. Each request on a connection is handled at a different OServ from the previous one (i.e. four requests R0–R3 are handled at OServ1, OServ2, OServ3, and OServ1 re-

spectively). Each request generates a response size of 15 KB that is sent back from the optimal servers directly to the client. Figure 8(a) shows the effect of multiple handoffs on connection lifetimes. The graph shows that even after multiple handoffs, the loss in response time per handoff remains same(i.e. the gap between the lines divided by the number of handoffs remains constant). This result demonstrates that half-pipe anchoring succeeds in keeping the handoff overhead low by (1) anchoring the control pipe and transferring little data between control and data pipes and (2) preventing data pipe draining through pipelined request processing.

5.4. Performance Comparison to KNITS

To the best of our knowledge, KNITS [27] is the only other implementation available today that supports multiple connection handoffs. We compared the performance of Split-stack with that of KNITS using previously published data [27]. Figure 8(b) compares the overhead in connection lifetimes that Split-stack incurs with the overhead that KNITS incurs as the response size is varied. Both overheads are normalized to the case with no handoff (the "direct" case). The figure shows that the amount of overhead we incur is about one-fourth that incurred by KNITS. Another observation in the graph is that, while the overhead for KNITS remains almost the same, Split-stack's overhead decreases significantly as the response file size increases. For example, between a file size of 1.5 KB and 24 KB, the percentage overhead of Split-stack falls down by 50%, while that of KNITS decreases by just 2%. Split-stack's improved performance can be explained by the fact that it incurs an overhead only during the data pipe setup phase. In contrast, KNITS has two additional sources of overhead: it must forward all response packets through the frontend, and perform address translation on the packets. This result shows that backend based handoff mechanisms are more efficient than frontend based mechanisms.

6. Conclusions

We present the design of a multiple connection hand-off mechanism based on half-pipe anchoring. Our work is motivated by the changing trends in content server architectures. The key insight behind half-pipe anchoring is to decouple the two unidirectional half-pipes that make up a TCP connection. This technique anchors the control pipe at a designated server while allowing the data pipe to migrate on a per-request basis to the server best suited to service the request. We have shown a simple design and implementation of a multiple handoff mechanism based on half-pipe anchoring. Our performance analysis shows that our technique incurs low overhead; furthermore, because it is back-end based, it is highly scalable. We compared our prototype implementation to KNITS, an existing system that supports multiple handoffs, and found that our implementation incurs at most one-fourth of the overhead that KNITS incurs. We believe that one of the most interesting features of half-pipe anchoring is to enable the building of heterogeneous server clusters that are highly scalable and energy efficient.

7. Acknowledgments

We thank Eric Van Hensbergen of IBM, Austin for providing us the data related to KNITS.

References

[1] Apache. http://www.apache.org.
[2] NISTNet network emulator. http://snad.ncsl.nist.gov/itg/nistnet/.
[3] Omnicluster Technologies Inc., Boca Raton, Florida. http://www.omnicluster.com/Omni2FINAL.pdf.
[4] RedHat Inc. TUX 2.1. http://www.redhat.com/docs/manuals/tux/.
[5] Redline Networks. http://www.redlinenetworks.com.
[6] Resonate Dispatch. http://www.resonateinc.com.
[7] RFC 1323: TCP Extentions for high performance.
[8] RFC 2018: TCP Selective Acknowledgement Options.
[9] RFC 2616: Hypertext Transfer Protocol – HTTP/1.1.
[10] RFC 793: Transmission control protocol.
[11] RLX Technologies Inc., The Woodlands, Texas. http://www.rlx.com/product/features/server_blade.html.
[12] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for P-HTTP in cluster-based web servers. *In Usenix Annual Technical Conference*, June 1999.
[13] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution cluster-based network servers. *In Usenix Annual Technical Conference*, 2000.
[14] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. Tcp behavior of a busy internet server: Analysis and improvements. *In IEEE INFOCOMM*, March 1998.
[15] P. Bohrer, E. Elnozahy, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. The case for power management in web servers. In *Power Aware Computing, Kluwer Academic Publications*, 2002.
[16] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *18th Symposium on Operating System Principles*, 2001.
[17] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for url-aware redirection. *In Usenix Annual Technical Conference, San Diego, CA.*, June 2000.
[18] P. Fox. Dynamic web pages prepared for takeoff. http://www.computerworld.com/softwaretopics/erp/story/0,10801,67114,00.html, Jan 2002.
[19] G. Govatos. Speeding up your dynamic content. http://www.nwfusion.com/news/tech/2000/1218tech.html.
[20] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. IBM TechReport, May 1997.
[21] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High performance web site design techniques. In *Proc. of IEEE Internet Computing, Volume: 4 Issue: 2, pp. 17-26*, 2000.
[22] R. Kokku, R. Rajamony, L. Alvisi, and H. Vin. Issues in improving web server performance. *In 1st Austin CAS conference, IBM, Austin*, July 2000.
[23] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias. Design and preformance of a web server accelerator. *In IEEE INFOCOMM*, 1999.
[24] D. Mosberger and T. Jin. Httperf – a tool for measuring web server performance. In *In SIGMETRICS Workshop on Internet Server Performance.*, 1998.
[25] V. Padmanabhan and J. Mogul. Improving HTTP latency. *In 2nd International WWW Conference, Chicago, IL*, Oct 1994.
[26] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. *In ASPLOS*, 1998.
[27] A. Papathanasiou and E. V. Hensbergen. KNITS: switch-based connection handoff. In *In IEEE Infocom.*, 2002.
[28] S. Puglia, R. Carter, and R. Jain. MultECommerce: A distributed architecture for collaborative shopping on the www. In *ACM Conference on Electronic commerce*, 2000.
[29] M. Rangarajan, A. Bohra, K. Banerjee, E. Carrera, and R. Bianchini. TCP servers: Offloading TCP processing in internet servers. design, implementation and performance. Technical report, Rutgers University, 2002.
[30] H. V. Shah, D. B. Minturn, A. Foong, G. L. McAlpine, R. S. Madukkarumukumana, and G. J. Regnier. CSP: A novel system architecture for scalable internet and communication services. In *USITS*, 2001.
[31] A. Snoeren, D. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *USITS*, 2001.
[32] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available internet services using connection migration. In *ICDCS*, 2002.
[33] W. Tang, L. Cherkasova, L. Russell, and M. Mutka. Modular TCP handoff design in streams based TCP/IP implementation. In *1st International Conference on Networking.*, 2001.
[34] H. Zhu, T. Yang, Q. Zheng, D. Watson, O. H. Ibarra, and T. R. Smith. Adaptive load sharing for clustered digital library servers. *Int. J. on Digital Libraries*, 2000.