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 novel technique to build a multiple connection handoff mechanism that enables efficient use of resources in a server cluster, improves the scalability of the cluster and supports construction of heterogeneous cluster architectures where nodes are specialized to efficiently perform specific tasks of client requests. 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 server in the cluster and 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 prototype multiple connection handoff mechanism in the Linux kernel and demonstrate 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. A recent study conducted by the HTRC group [17, 18] reveals 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 [20, 32]. Node specialization improves efficiency and scalability of the cluster and reduces overall energy consumption [14, 15, 2, 10]. 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 [3], while dynamic requests that typically require complex cookie parsing or database accesses are better handled by a user level server application like Apache. Nodes can also be specialized to perform specific parts of request processing efficiently [27, 28]: 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 [26] specialize functionalities performed by nodes to achieve scalable systems.

Third, clients can send multiple, and potentially very different, requests to the same server using a single persistent TCP connection [8, 23]. 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*.

*This work has been supported by an IBM Co-operative Fellowship for Ravi Kokku and an IBM Faculty Partnership Award for Lorenzo Alvisi and Harrick Vin. This work was initiated during a Research Internship at the IBM ARL during the Summer of 2000.

Half-pipe anchoring goes beyond existing connection handoff protocols [4, 5, 19, 24, 25, 29, 30, 31] in at least two respects. First, it supports efficient *multiple* handoffs of the same connection by allowing requests 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, our 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 against an existing multiple handoff solution(KNITS) [25] and find that the overhead incurred by our solution is only one-fourth that of KNITS in the worst case.

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 allowing a request to be serviced by a server best-suited to service the request can be addressed using two end-to-end approaches. In the first approach, content and services can be authored such that requests can be sent directly from the client to the appropriate servers. This approach has a drawback that it requires a client to open multiple connections with servers within a server cluster that wastes both server and network resources [13, 23]. Also, this approach exposes the configuration details of the server cluster to the content authoring process (and hence the clients) and prevents the cluster from achieving fine-grained load balancing across servers.

The second end-to-end approach uses a transport-level connection migration mechanism [29, 30] to migrate a transport connection between a client and a server to another server better-suited to service the next request. The client application is transparent to this migration. This approach incurs large connection migration latencies because of WAN delays and hence is not suitable for migration between short web transfers. Also, this approach needs client system modifications limiting its deployability.

Because of the drawbacks of the end-to-end approaches, we focus on *cluster-based* mechanisms to ensure that each request is serviced by the best-suited server. 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 [24]. The backend servers service client requests. The frontend and the backend servers are connected using an internal high-speed network. We refer to a backend server that holds the connection from a client as the *designated* server, and the backend server that is best-suited to service the client request as the *optimal* server.

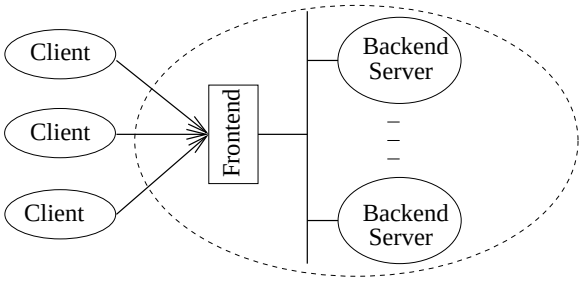


Figure 1. A simplified cluster architecture

To classify and systematically explore 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 as coming from the frontend node. This modification ensures that 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 re-

source utilization, and (3) other constraints imposed by the approaches on the cluster architecture.

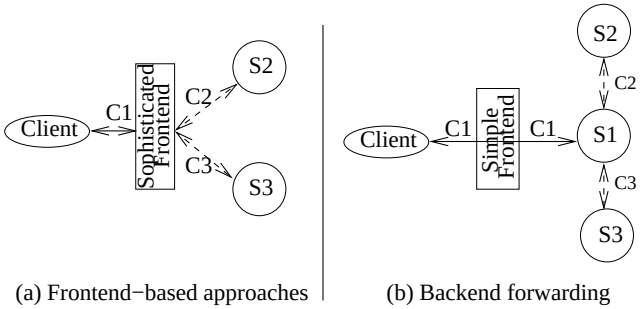


Figure 2. Cluster based mechanisms

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 (i.e., determines the optimal server 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 [4] fall into this category.

The main advantage of this approach is that all the above three components are encapsulated in the frontend. 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 the frontend node becomes a bottleneck and limits the overall scalability of the server cluster. Although techniques such as TCP splicing [16] 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 [12].

Hybrid Approaches These solutions split the three components between the frontend node and the backend servers. KNITS [25] is an instance of this approach. In KNITS, layer-7 switching is done at the backend servers, while the frontend node is responsible for cluster transparency and connection management. 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 overhead involved in manipulating response packets at the frontend 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 [12].

The simplest instantiation of the backend approach is often 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 persistent connections (C2 and 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, 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.

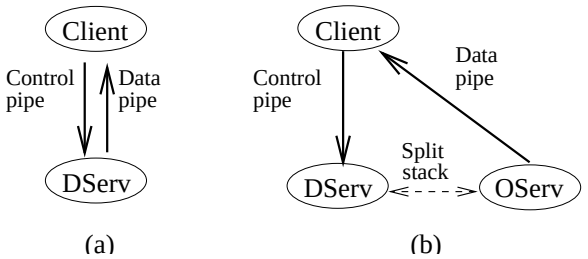


Figure 3. Separating the two half–pipes

3. Half-pipe Anchoring

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.

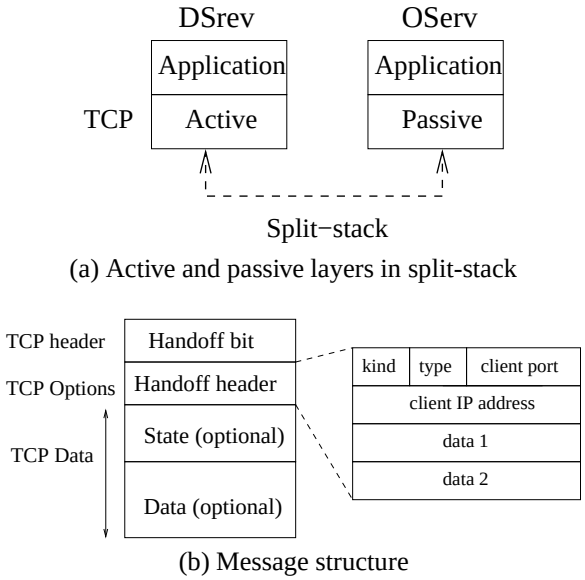
Multiple Connection Handoffs Consider two successive requests, r_1 and r_2, received by DServ on the same persistent connection from a client. Two scenarios can occur: (1) r_2 reaches DServ after the client has completely acknowledged the response to r_1, or (2) r_2 reaches DServ while r_1 is still being 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 [11].

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 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 [9]). 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 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 handoff instance:

<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 straight forward 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 waiting on OServ-port.

3. Cluster transparency: When the OServ Appl sends data to the passive layer to be sent to the client, the passive layer modifies the sender address in the TCP header to indicate that the data actually originated from the designated server.

4. SS_DONE and SS_FAILURE:

The OServ Appl closes the connection as soon as the response data is sent to the passive layer. The passive layer, on receiving the close, does not trigger the normal TCP-FIN exchange [9] with the client and instead returns success to the OServ Appl.

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 application exists waiting on the given port 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 acknowledging data that is potentially sent out by this passive, then the active still sends a control message to the passive.

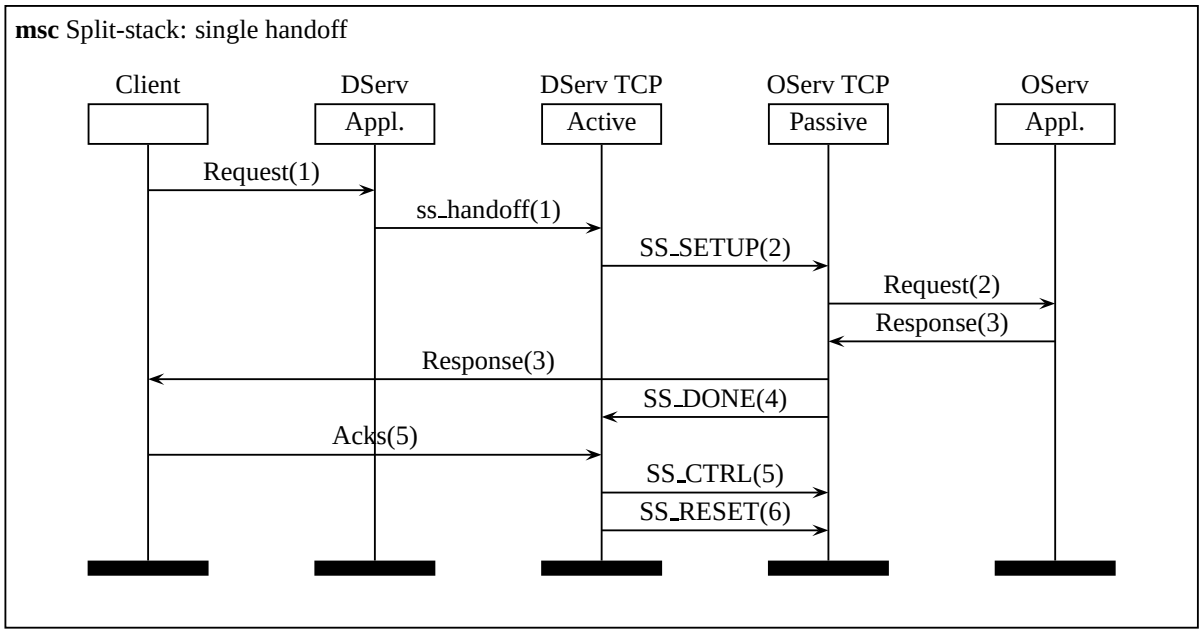


Figure 5. Message sequence chart for single connection handoff using split-stack

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 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

If a new request arrives at the designated server after the response for the previous request is completely acknowledged by the client, and the new request also requires a handoff, the handoff can happen without any more changes to the protocol described in Section 4.2.

Now consider a case where a new request arrives while a handoff for another request is already in progress. In order to minimize the request processing latency, ideally the request should get processed as soon as the DServ Appl processes the request. However, the earliest time at which an active layer can initiate a second handoff is after it receives the SS_DONE message from the first passive layer where the first handoff initiated a data pipe. This is when the active layer has complete state information of the connection. In our prototype, we allow initiation of handoff for the next request as soon as SS_DONE is received for the previous request. This pipelined processing of requests 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.

Our approach to handling correct forwarding of acks and request data is based on the observation that an ack actually 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 (and queued to handle retransmissions) has been received at the client and hence the space occupied by the queued packet can 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 be performed always.

Two facts enable us to determine what to do with an ack received at a designated server. First, at any given time, only one optimal server can be transmitting new data packets to the client, while multiple servers could be waiting for acks for the data they already sent. Second, when an optimal server has finished sending data to a client, it informs the designated server of the amount of data sent. Using this information and the ack sequence numbers, the designated server determines which server node should reclaim packets and which server node should send a new data packet. The active layer then sends SS_CTRL messages to the corresponding passives to either reclaim packets or send more packets or retransmit any lost packets. If all packets sent by an optimal server are acknowledged, then the active sends an SS_RESET message to its passive layer.

4.4. API

If a DServ Appl determines that a request has to be handled remotely, it executes 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. If not, 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 pre-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 as soon as the request is completely served. This triggers the SS_DONE message from the passive layer.

4.5. Discussion

Our prototype implementation currently makes some simplifying assumptions about TCP by not handling some TCP options (TCP timestamps [6] and SACKs [7]). Although these TCP options are important, they were not required to demonstrate the efficacy of half-pipe anchoring and hence we chose not to implement them.

Since the active layer completely controls the passive layer and since the designated server receives all the packets from the client, any information related to TCP options can be easily communicated to the passive layer by using the SS_CTRL message. For example, in order 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 [6]. To achieve this monotonicity, the designated server can send its present timestamp to the optimal server receiving credits to send new packets and the optimal server can use this timestamp in the response packets. SACK options can be interpreted and sent as retransmission requests in the SS_CTRL message. However, some options like Window scale and MSS that are exchanged during SYN exchange phase between the client and the designated server automatically get reflected in the connection state sent to an optimal server in the SS_SETUP message and hence do not require any special handling in our protocol.

One 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 Levy-Abegnoli et al. in [21] to accelerate web servers) that implements just the functions of a passive layer like sending more packets, retransmitting packets and reclaim resources on the directions of the active layer and thus improve the performance and scalability of the cluster.

With a minor extension to the SETUP_DONE message (allowing it to carry back data) our prototype can support dependent dynamic requests (two requests are dependent if the state modified by one request is used by the other). However, the DServ and OServ Appls 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 transferring 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 manage explicitly the state that they require to operate correctly.

5. Prototype Performance

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 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. 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 [22] to drive our experiments. We modified Httperf to maintain a specified number of outstanding connections to the server.

5.2. Benchmark Experiments

In the first set of experiments, we measure the various overheads of our implementation using setup1 of Figure 6. In the graph shown in Figure 7(a), the line “direct” corresponds to the experiment done with the vanilla 2.4.10 TCP/IP stack, and the line “direct with kernel mods” corresponds to the kernel with our modifications to the stack to implement the active and passive layer functionalities. For lines ”direct” and ”direct with kernel mods”, DServ sends a response directly to the client’s requests. Each connection setup with the client carries one request. The negligible spacing between “direct” and “direct with kernel mods” lines in figure 7(a) shows that our modifications to the kernel do not add much overhead to the normal message flow in the vanilla network stack.

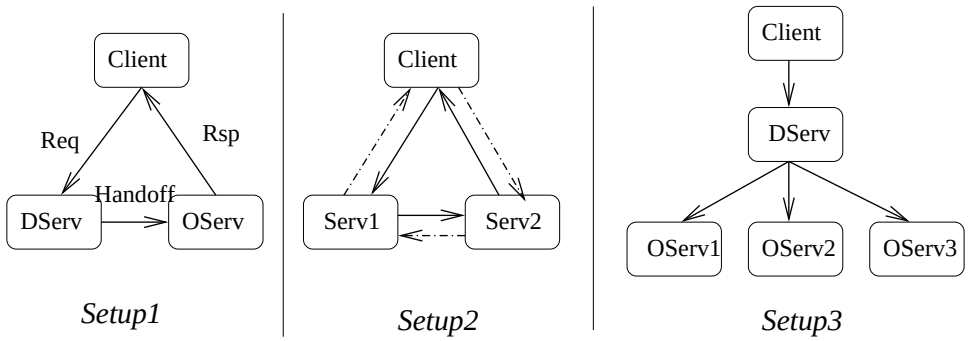
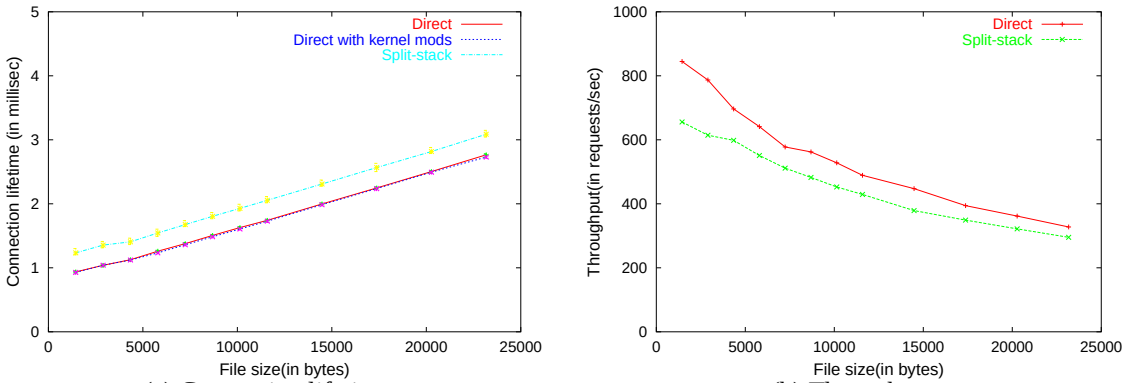


Figure 6. Experimental setup



(a) Connection lifetimes (b) Throughput

Figure 7. Effect of varying response size

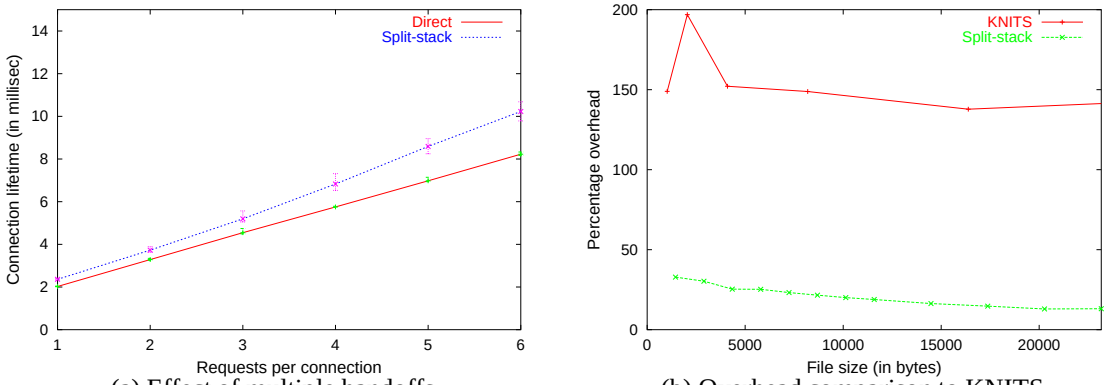
Figures 7(a) and 7(b) show the overhead of the split stack protocol by measuring the response time and throughput perceived by the client for one request on a connection as response size is varied. The "direct" line in the graph corresponds to the case where the client obtains a response directly from a server (DServ). The "Split-stack" line corresponds to the case where the client sends the request to DServ, which hands-off the connection to OServ to service the request. The gap between the two lines represents the overhead because of the split-stack protocol. The constant spacing between the two lines in Figure 7(a) indicates that most of the overhead is incurred in the initial setup. Note here that the overhead involves contribution from two applications in the split-stack case (DServ Appl and OServ Appl) as against only one application in the direct case (This setup is justified because any connection handoff mechanism always requires processing on both the nodes). Also, the exact overhead contributed by the application is dependent on the application itself.

Table 1 shows the connection lifetimes over a WAN as perceived by a client requesting a document of size 15 KB with varying WAN delays (where all request and response packets get delayed by a constant amount of time as shown on the X-axis). The difference in connection lifetimes remains constant irrespective of the WAN delays showing that most of the overhead is incurred in the handoff initiation. The data supports the fact that the handoff overhead becomes negligible in a WAN environment, where packet delays are at least a couple of orders of magnitude more than in a LAN environment. To simulate WAN delays, we used the NISTNet delay router [1].

Table 1. WAN delay *vs.* Connection lifetimes

WAN delay *(in ms)*	Direct *(in ms)*	Split-stack *(in ms)*	Overhead
2	12.23	12.73	4%
10	44.44	44.94	1.1%
20	84.47	85.08	0.72%
40	164.54	165.14	0.36%

Any handoff mechanism uses more resources than the direct case because of handoff initiation overhead, ack for-



(a) Effect of multiple handoffs

(b) Overhead comparison to KNITS

Figure 8.

warding and double application processing. To get a sense of how all these factors affected, we measured the throughput achieved by two servers with the setup2 of Figure 6). In the direct case, requests sent to the servers Serv1 and Serv2 from the client are processed locally by the servers. In the split-stack case, each server hands-off the request that it receives from the client to the other server, i.e. no request is processed locally. This is a worst-case setup and brings out the worst-case performance degradation of our technique.

Our measurements showed that in this setup, the direct case achieved a maximum throughput of 1348 reqs/sec, while split-stack case achieved 796 reqs/sec, which is a performance degradation of 40%. Most of the overhead is incurred due to double application processing that is inherent in any handoff mechanism. Using a customized server application that is optimized for request processing, instead of Apache on the designated server, could further reduce some of the overhead. This observation also suggests 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

In this experiment, we measured the efficacy of half-pipe anchoring in handling multiple handoffs. The setup used for this experiment is shown in setup3 of Figure 6. The 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 than the previous one (i.e. six requests means six handoffs on the connection). Each request generates a response size of 15 KB that is sent back from the optimal servers directly to the client. The graph shown in 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. This graph supports the fact that half-pipe anchoring successfully keeps the handoff overhead minimum by (1) anchoring the control pipe and transferring minimum data between control and data pipes and (2) allowing pipelining of request processing to prevent data pipe draining.

5.4. Performance Comparison to KNITS

To our knowledge, KNITS [25] is the only other implementation available today that supports multiple connection handoffs. We compared the performance of split-stack with that of KNITS using the data in [25]. Figure 8(b) compares the overhead in connection lifetimes that split-stack incurs as compared to the direct case, with the overhead that KNITS incurs as compared to the direct case as response size is varied. The figure shows that the amount of overhead we incur is only one-fourth that incurred by KNITS even in the worst case (with a 1.5 KB response). Another observation in the graph is that, while the overhead for KNITS remains almost the same, the overhead of split-stack falls down significantly as 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 the overhead of KNITS falls by just 2%. This is explained by the fact that the overhead in split-stack is incurred only during the data pipe setup phase. Whereas, the overhead in KNITS is due to the handoff and also due to the overhead of forwarding all response packets through the frontend, while performing address translation on the packets. This experiment supports that backend based handoff mechanisms are more efficient than frontend based mechanisms. We could not compare our throughput with that achieved by KNITS for lack of enough data related to KNITS. However, we believe that the throughput comparison would match that of the overhead comparison, since the more overhead a system has, the less throughput it achieves.

6. Conclusions

We present the design of a multiple connection handoff 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 backend 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] NISTNet network emulator. http://snad.ncsl.nist.gov/itg/nistnet/.

[2] Omnicluster Technologies Inc., Boca Raton, Florida. http://www.omnicluster.com/Omni2FINAL.pdf.

[3] RedHat Inc. TUX 2.1. http://www.redhat.com/docs/manuals/tux/.

[4] Redline Networks. http://www.redlinenetworks.com.

[5] Resonate Dispatch. http://www.resonateinc.com.

[6] RFC 1323: TCP Extentions for high performance.

[7] RFC 2018: TCP Selective Acknowledgement Options.

[8] RFC 2616: Hypertext Transfer Protocol – HTTP/1.1.

[9] RFC 793: Transmission control protocol.

[10] RLX Technologies Inc., The Woodlands, Texas. http://www.rlx.com/product/features/server_blade.html.

[11] M. Aron, P. Druschel, and W. Zwaenepoel. Efficient support for P-HTTP in cluster-based web servers. *In Usenix Annual Technical Conference*, June 1999.

[12] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution cluster-based network servers. *In Usenix Annual Technical Conference*, 2000.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] P. Fox. Dynamic web pages prepared for takeoff. http://www.computerworld.com/softwaretopics/erp/story/0,10801,67114,00.html, Jan 2002.

[18] G. Govatos. Speeding up your dynamic content. http://www.nwfusion.com/news/tech/2000/1218tech.html.

[19] G. Hunt, E. Nahum, and J. Tracey. Enabling content-based load distribution for scalable services. IBM TechReport, May 1997.

[20] 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.

[21] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias. Design and preformance of a web server accelerator. *In IEEE INFOCOMM*, 1999.

[22] D. Mosberger and T. Jin. Httperf – a tool for measuring web server performance. In *In SIGMETRICS Workshop on Internet Server Performance.*, 1998.

[23] V. Padmanabhan and J. Mogul. Improving HTTP latency. *In 2nd International WWW Conference, Chicago, IL*, Oct 1994.

[24] 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.

[25] A. Papathanasiou and E. V. Hensbergen. KNITS: switch-based connection handoff. In *In IEEE Infocom.*, 2002.

[26] S. Puglia, R. Carter, and R. Jain. MultECommerce: A distributed architecture for collaborative shopping on the www. In *ACM Conference on Electronic commerce*, 2000.

[27] 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.

[28] 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.

[29] A. Snoeren, D. Andersen, and H. Balakrishnan. Fine-grained failover using connection migration. In *USITS*, 2001.

[30] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Highly available internet services using connection migration. In *ICDCS*, 2002.

[31] 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.

[32] 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.