

Half-Pipe Anchoring: An Efficient Mechanism for TCP Connection Handoff

Ravi Kokku, Ramakrishnan Rajamony, Lorenzo Alvisi, Harrick Vin
IBM Technical Contact: Ramakrishnan Rajamony, Austin Research Laboratory

Abstract—We present the design and implementation of Split-Stack, a novel and efficient mechanism for handing off TCP connections that enables fine-grain load balancing among the servers of a cluster. The key insight behind our approach is to decouple the two unidirectional half-pipes that make up a TCP connection. Our approach anchors 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 the server best suited to service the request. This approach has two benefits. First, it permits the half-pipe from the cluster to be utilized to the full extent of its bandwidth. Second, it permits all client requests to be intercepted at a single designated server, facilitating the construction of heterogenous systems out of modularized components. We demonstrate these benefits by applying the Split-Stack approach to the problem of designing scalable web servers capable of supporting persistent HTTP. Our approach yields upto 65% higher throughput than alternative schemes for fine-grained load balancing.

Keywords— Web server performance, Network system software, Network protocols

I. INTRODUCTION

Server clusters are behind most web services employed on the Internet today. Clusters are attractive because they combine the price-performance characteristics of commodity hardware with the ability to increase server throughput by adding more server nodes to the cluster. A fundamental problem that these systems need to address is scalability: ideally, the number of client requests that a cluster can service should grow linearly with number of servers.

Several load-balancing techniques—round-robin, content-based, server-load-based, etc.—have been

R. Rajamony is with the Systems Software Department, IBM Research, Austin

R. Kokku, L. Alvisi, and H. Vin are with the Univ. of Texas, Austin

proposed to address this problem [1]. These techniques work well if each request is received over a separate TCP connection, as in HTTP1.0 [2]. However, the overhead involved in setting up and tearing down a connection for each request hurts scalability [3], [4], and increases the latency of a server’s response. To reduce this overhead, HTTP/1.1 allows clients to send multiple HTTP requests to a server through a single TCP connection [5]. By making connections persistent, HTTP/1.1 amortizes TCP establishment and tear down costs and increases network utilization by avoiding multiple TCP slow starts. Unfortunately, persistent connections can limit the effectiveness of load balancing schemes. When a persistent TCP connection is established, the load balancing mechanism selects a server to handle all requests arriving on the connection (we call this server the connection’s *designated server*, or DS)¹. Suppose now that the entity initiating a persistent connection with the cluster is a proxy server. Since a proxy server multiplexes requests from multiple clients onto the same connection, it is quite likely that different servers within the cluster will be best suited to service different requests. For instance, if the proxy is connected to a cluster that uses a content-based load balancing scheme, the server whose cache is best equipped to service requests coming from proxy client c_1 may not have cached the data required to respond to unrelated requests coming from proxy client c_2 . In other words, persistent connections can prevent load-balancing schemes from directing a given request to its *optimal server* (OS), i.e. the server that is best

¹In this paper we explicitly abstract away the details of the implementation of the load balancer and of the mechanism used by the load balancer to select the designated server. This mechanism may consist of a simple distributor that implementing a round robin policy [6], or of a more sophisticated protocol, that may itself involve TCP connection handoffs [1], [7]. We factor these details out of our study to isolate the effects of the policy that is chosen, once a DS has been selected, to restore fine grain load balancing over persistent connections.

suites to service it: this coarse-grain load balancing limits the scalability of the cluster.

There are two basic approaches to restoring finer-grain load balancing. In the first approach DS forwards each request to the appropriate OS, which services the request and then forwards the results back to DS; DS then returns the results back to the client. In the second approach, when DS receives on a connection a request for which it is not the optimal server, it not only forwards the request to the appropriate OS, but with the request transfers to the OS also the responsibility of handling any successive requests coming on that connection. The OS forwards the results directly to the client, bypassing the DS. All successive TCP packets from the client are sent through DS to OS: in effect, OS becomes the connection's new DS, while the original DS becomes a router on the path that connects the client to the new DS. The first approach is generally known as *back-end forwarding*, while the second is known as *connection handoff* [1]. Connection handoff is generally acknowledged in the literature as providing somewhat better performance than back-end forwarding, but at the cost of a much higher implementation complexity [1], [6], especially if, during its lifetime, a connection is handed off multiple times to different servers. Prior studies have typically concluded that the performance improvements yielded by connection handoff do not justify the increased implementation complexity.

In this paper, we argue that the news of connection handoff's demise may have been greatly exaggerated. First, we show that in several system configurations connection handoff can outperform significantly back-end forwarding. Second, we present the first design and implementation of a connection handoff protocol for web server clusters. Our approach departs radically from the conventional architectures proposed for performing connection handoff in other settings, such as mobile computing [8]. In our architecture, what is handed off to an OS is not the entire connection, but only the responsibility of servicing and responding to the requests for which OS is the optimal server. Our approach builds on the observation that a client's TCP connection to the cluster consists of two half pipes, one from the client to the cluster and the other from the cluster to the client; this leads us to propose an architecture in which the first half pipe is anchored to the

DS for the duration of the connection, while the second half pipe is transferred on a per-request basis to the appropriate OS. Third, we show that our approach is efficient; in particular, we show that our architecture yields an upto 60% improvement in server throughput over conventional techniques for implementing multiple connection handoffs. Finally, we observe that our architecture introduces asymmetry in the capabilities required of DSs and OSs; DSs can be optimized for HTTP and TCP processing while OSs can be tuned for efficient data delivery. This facilitates the construction of scalable servers from heterogeneous components (e.g., by combining TCP accelerators with network attached storage (NAS) appliances).

The rest of the paper is organized as follows. Section II describes connection handoff and the implementation challenges involved. Section III describes the core idea of this paper: anchoring a connection half-pipe. Section IV presents Split-Stack, our implementation of the connection handoff protocol described in Section III. We evaluate our Split-Stack implementation in Section V, present related work in Section VI, and offer our concluding remarks in Section VII.

II. UNDERSTANDING CONNECTION HANDOFF

Although this paper is the first to present an actual implementation of a complete connection handoff protocol for web servers, it is not the first paper to consider the possibility of such an implementation [1]. Previous studies have argued against using connection handoff on two grounds. First, in simulations connection handoff appears to yield small performance gains over back-end forwarding. Second, any actual implementation of connection handoff that attempts to reap these gains faces significant technical challenges. In this section, we revisit these issues.

A. Performance

Aron et al [1] compare the effectiveness of back-end forwarding and connection handoff as mechanisms to achieve fine grained load balancing over persistent HTTP connections; in particular, they consider cluster-based web servers that use their locality-aware request distribution protocol. The conclusion of their simulations is that, although connection handoff provides slightly higher throughput than back-end forwarding,

the percentage-wise performance difference between the two approaches is less than 6%. We believe that these conclusions, although accurate for the simulation setup considered in [1] convey only part of the story.

The performance of a connection handoff implementation depends on multiple parameters, including the hardware configuration of the cluster’s servers (e.g., the amount of their cache memory, their disk bandwidths, the number of processors in the cluster, etc.), the size of the data set, the ratio between requests for static and dynamic content, and the percentage of requests received on a connection that cannot be optimally serviced at the current designated server. The experimental evaluation in [1] explores a specific point in the multi-dimensional space defined by these parameters: it considers a workload generated by combining logs from multiple departmental web servers at Rice University, a data set of about 1 GBytes, and a cluster consisting of 1 to 10 servers, each with 85MBytes of cache.

As Figure 1 illustrates, exploring different points in the parameter space may lead to very different conclusions. The figure shows the result of simulating connection handoff and back-end forwarding in cluster-based web servers that implement the locality-aware request distribution protocol [1]. We use a trace-driven workload obtained from IBM’s Olympics 98 web site, with a data set size of about 1.1 GBytes. The figure shows three curves; each curve represents, for a specific server configuration (in terms of the amount of available cache at each server), the effect of changing the number of servers in the cluster on the improvement in performance yielded by the connection handoff over the back-end forwarding. Note that, for clusters of up to 15 servers, the 85 MBytes curve shows connection handoff delivering between 20% and 30% more throughput than back-end forwarding, as opposed to the 6% or less reported for the same cache size and a similar data-set size in [1]. The reason for the discrepancy is in the nature of the workloads used in the two simulation studies. The Olympics workload exhibits a much higher number of request per connection than the Rice University trace, and, within a connection, a significantly higher percentage of *unrelated requests* (i.e. request that cannot be served optimally at the designated server). Indeed, we have performed

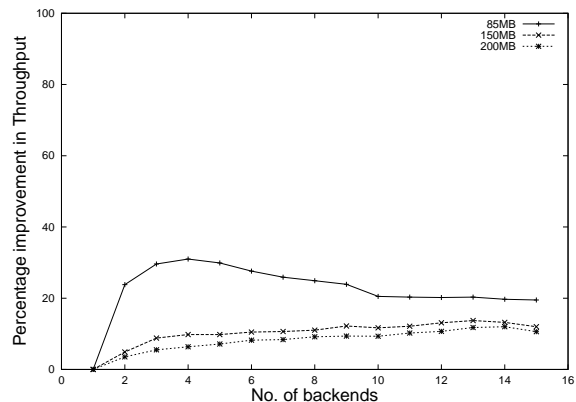


Fig. 1. An analysis of performance of connection handoff and backend forwarding

other simulations that show that when we substitute the Olympics trace with a trace obtained by combining logs of departmental web servers at UT Austin, we obtain results in line with those reported by [1].

Figure 1 also illustrates that the relative performance of connection handoff depends heavily on the relationship between the amount of cache memory available at a server and the size of the data set. For a given data-set size, the larger the cache at each server, the smaller the benefits of connection handoff over back-end forwarding. In fact, if the amount of cache available at a server were to exceed the size of the data set, there would be only negligible gain in using connection handoff, since the designated server would eventually be capable of becoming the optimal server for any request.

Interestingly however, a new set of upcoming server designs that target not just raw performance but also spatial density and energy consumption advocate architectures that would have clusters operate in the area of the above-mentioned parameter space in which connection handoff yields significantly more throughput. The OmniCluster SlotServer 1000 is an example of these kind of architectures[9]. The SlotServer has an x86 compatible processor and can hold between 32MB and 256MB of memory. The SlotServer is intended for use as proxy and content servers. The RLX ServerBlade design from RLX Technologies Inc. [10] is another example of a web server design that looks beyond pure performance. The ServerBlade is powered by a Transmeta Crusoe TM5600 processor and can support between 128MB and 512MB of memory.

In summary, we believe that our experiments, though not exhaustive, illustrate that connection handoff *can* result in performance gains significant enough to justify investigating the technical challenges that complicate its implementation.

B. Implementation Challenges

As we pointed out above, the literature on server clusters does not contain any actual implementation or detailed design description of a connection handoff protocol capable of supporting multiple handoffs. This problem however has been addressed in the context of mobile computing with the objective of keeping alive the TCP connections of a mobile host while it moves between cells. Two implementations of interest in this respect are I-TCP [8], and ALICE [11].

In these schemes, a connection is handed off from one server to another only after the first server has received all of the acknowledgements for its packets. This approach simplifies the implementation of connection handoff, since at any time, the TCP acknowledgements and requests need to be processed at exactly one server); however, it has the unfortunate side effect of draining the TCP data pipe completely before a connection is handed off to another server. A second problem that these implementations exhibit is the inefficiency that results when, as a side-effect of multiple connection handoffs, the path from a client to the current designated server has to traverse all the former designated servers for that connection (connection chaining) [11].

These problems, especially the draining of the TCP pipe, have been mentioned [1] has potential show-stoppers towards an efficient implementation of connection handoff. In the next section, we present an architecture that addresses these problems.

III. ANCHORING A CONNECTION HALF-PIPE

A. Key Idea

The connection handoff paradigm is predicated on the premise that multiple requests from a client sent over the same connection are best serviced by different nodes of the server cluster. For instance, in the context of a web server cluster, client requests for static data may be best serviced by nodes different than those that service requests for dynamically generated data. Thus, in the connection handoff paradigm, each client

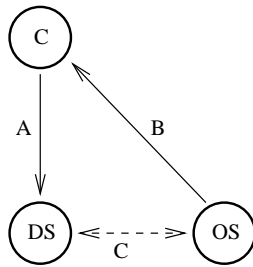
request on a connection may have a distinct *optimal server* (OS) where it is best serviced.

The bidirectional TCP connection over which client requests and server responses travel is actually made up of two unidirectional *half-pipes*. The half-pipe from the client is used to send requests to the server. The half-pipe from the cluster is used to send responses from the server. At a connection endpoint, TCP performs flow control and reliability actions on one half-pipe based on protocol messages (acknowledgements) it receives from the other half-pipe. Traditionally, the two half-pipes of a TCP connection are tightly bound together and reside on the same node; in fact, no other alternative makes sense on a TCP connection between a single client and a single server.

In the abstract, the requirement that the two half-pipes of a TCP connection be tightly bound together can be relaxed as long as the flow control and reliability actions across the half-pipes are maintained. Our key insight is that in order to support connection handoff, while the half-pipe from the cluster needs to move from one optimal server to another depending on client requests, the half-pipe from the client can remain stationary. Specifically, the half-pipe from the client can be anchored at a single *designated server* (DS) for the duration of the connection.

Anchoring the half-pipe from the client at a single designated server has two important benefits. First, anchoring permits the half-pipe from the cluster to be utilized to the full extent of its bandwidth. Second, anchoring permits all client requests to be intercepted at a single designated server, facilitating the construction of heterogeneous systems out of modularized components. We elaborate on these benefits in greater detail in Section III-C.

Figure 2 illustrates the notion of separating the two connection half-pipes. In the figure, a client has opened a connection to a server. The half-pipe from the client is anchored at a designated server. An application executing on the designated server first pre-processes incoming client requests and determines the optimal server where the request is best serviced. If the optimal server is the designated server itself, the request is serviced locally. Otherwise, the DS instantiates a half-pipe to the client at the optimal server, from where the response is directly sent to the client bypassing the DS. The role of the designated server is to con-



A: Half pipe from client
 B: Half pipe to client
 C: Split-stack protocol

Fig. 2. Separating the two half-pipes of a connection

control and co-ordinate the movement of the half-pipe to the client among the different optimal servers for the requests on that connection. Flow control and reliability are maintained using a lightweight communication protocol between the designated and optimal servers.

In a homogeneous cluster, the roles of designated server and optimal server could change on a per-connection basis. In fact, what is a designated server for one connection may very well be the optimal server for a request on a different connection.

B. The Split-Stack Protocol

We have designed a connection handoff protocol based on the principle of anchoring the connection half-pipe originating from the client. An application writer uses a system call we provide to handoff the half-pipe to the client between different server nodes. However, while the half-pipe from the cluster moves about from node to node, we maintain connection state centrally at the designated server for the connection.

B.1 The Application Perspective

Our connection handoff protocol is exposed to an application solely in the form of a new system call: `handoff(socket, request, OS, OS-port)`

An application executing on the designated server (termed the DS application) accepts an incoming connection as it would in any server system. The DS application has a choice after pre-processing client data received from the socket. It can either service the request itself, or it can use the handoff system call to request another application executing on a different

server node to service the request. A successful return from the handoff call tells the DS application that the request has been serviced by the remote application. It then proceeds to receive and pre-process the next request from the client.

An application executing on the optimal server (termed the OS application) also accepts incoming connections as it would in any server system. The OS application simply receives a request on this connection, services it by generating a response, and sends the response back on the connection. The optimal server transmits the response data directly to the client by-bypassing the designated server.

Figure 3 shows the pseudo-code for a DS and an OS application that together implement a higher-level function such as a clustered web server.

B.2 Protocol Overview

Separating the two-half pipes making up the connection requires us to translate feedback (protocol information) from one half-pipe into actions on the other half-pipe. We provide a high level description of how exactly our connection handoff protocol achieves this goal.

Consider a client that opens a connection to an application on the designated server (DS) using the TCP three-way handshake. After the connection is opened, the client sends a request to the DS application that pre-processes the request and determines the optimal server (OS) which must service the request. Normal TCP processing on the designated server acknowledges data sent by the client.

After determining the optimal server for a request, the DS application next initiates a connection handoff to the OS application. The handoff is achieved through the blocking `handoff(socket, request, OS, OS-port)` system call. The handoff call returns either after the OS application successfully sends out response packets to the client, or after a failure because of timeout or other error conditions.

Until this point, the designated server has control of the half-pipe from the cluster. As a result of the handoff system call, the designated server creates an instance of the half-pipe from the cluster on the optimal server. The half-pipe is instantiated on the optimal server by sending a setup message that communicates the TCP connection state, data for the OS application,

```

while(1) {
nc = accept (incoming conn.)
bytes = receive (nc, request, ...)
pre-process request
if(service request locally) {
    generate response
    send (nc,response)
}
else {
    OS = optimal server to service request
    OS-port = port where application on OS
                is listening
    handoff (nc, request, OS, OS-port)
}
}

```

```

hc = accept (incoming conn.)
bytes = receive (hc, request)
service request
generate response
send (hc, response)
close(hc)

```

Fig. 3. Pseudo-code for DS and OS applications that together implement a web server functionality

and the OS-side port where the OS application is listening.

On the optimal server, receipt of the setup message results in the creation of a socket with an established connection to the client. The connection is established instantaneously, *without* going through the normal three-way TCP handshake. The data sent by the DS application is provided to the OS application through the newly created socket. The OS application receives the data on the socket, services the request, generates a response to be communicated to the client. The TCP protocol stack on the optimal server sends this data to the client.

Meanwhile, acknowledgements sent by the client are received by the designated server. Since the client may be acknowledging data sent by a different server node than the designated server, the first step is to determine whom the acknowledgement is meant for. We describe acknowledgement processing in section III-B.3. Any incoming data from the client is passed on straight to the DS application.

When the OS application finishes sending its response to the client, it closes its socket to the client. This socket close is intercepted on the optimal server and the three-way TCP connection close handshake is *suppressed*. Instead, the optimal server simply informs

the designated server that the OS application has finished sending data. At this point, the designated server unblocks the DS application from its handoff call and is ready to move the half-pipe from the cluster to a different optimal server.

Note that the DS application can return from the handoff system call it makes only after the optimal server it specifies has sent *all* the response data to the client. This ensures that barring packet retransmissions, data packets bound for the client leave the cluster in-order.

The half-pipe from the cluster is moved about until the DS application decides to close it. The designated server then closes the half-pipe to the client, ending the connection.

B.3 Acknowledgement Processing

A TCP acknowledgement serves two purposes that are piggybacked onto one packet. First, it serves as a *credit* to send another packet on the connection. Second, it informs the sender that a previously sent packet can be *garbage collected* since it has reached the client. The need to garbage collect arises because TCP preserves unacknowledged packets in case they need to be retransmitted. While credits to send more packets are meaningful only when there is more data

to send, garbage collection is an action that always needs to be performed. Distinguishing between these two purposes is important since we allow the half-pipe from the cluster to move from node to node.

We process all client acknowledgements at the designated server. Two facts enable us to determine what to do with an acknowledgement. First, at any given time, only one server node in the cluster can be transmitting new data packets to the client. 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, the designated server can use the acknowledgement sequence number to find out which server node should garbage collect a previously sent packet, and which server node should send a new data packet.

C. The Benefits of Anchoring

C.1 Preventing the Data Pipe from Draining

TCP has a sophisticated mechanism to maximize bandwidth utilization between a server and a client even in the presence of large latencies. The TCP sliding window protocol permits a sender to transmit multiple packets before it stops and waits for an acknowledgement [12]. Sending new packets before a prior packet is acknowledged keeps the TCP data pipeline filled with packets. However, in practice, transient resource constraints can cause both data packets and acknowledgements to be dropped. TCP copes with such congestion by shrinking the maximum number of outstanding unacknowledged packets. Even under normal operation, packet losses can partially or even fully drain the TCP data pipeline.

Data pipeline draining can occur in the connection handoff paradigm for an entirely different reason. A naive handoff solution might hand off the connection from one node to another only after *all* outstanding packets sent by the first node have been acknowledged. Such a handoff solution would deliberately drain the data pipeline before handing the connection. In fact, Aron et. al. have acknowledged pipeline draining as a potential problem that a connection handoff protocol must address [7].

Our connection handoff protocol prevents data pipeline drains during connection handoff. As soon as the optimal server finishes sending the last data packet, it informs the designated server that it is done. At this

point, while the data packets from the first OS are still in flight, the designated server can direct a second optimal server to service a new request. The data pipe is thus prevented from draining. Anchoring the half-pipe from the client enables the designated server to forward client acknowledgements to the appropriate optimal server.

C.2 Support for Heterogeneity

Anchoring the half-pipe from the client at one server node for the connection duration facilitates building heterogeneous server clusters where different nodes are optimized to perform different functions. For instance, anchoring enables building a server cluster where the nodes are divided into different sets that perform combinations of (a) TCP connection state processing, (b) HTTP request header processing, (c) HTTP response header generation for static data, (d) static data serving, and (e) dynamic content generation.

Consider a static-data web serving cluster, where the parsing of client requests and generation of HTTP response headers occurs on one node, while response data is delivered from another server node. The DS and OS applications would be built along the lines shown in Figure 3. Instead of ever processing requests locally, the DS application always sends the HTTP response header to the client and has the OS application deliver the data.

Several researchers have advocated building server clusters using such a modular approach. The CSP architecture [13] consists of multiple functional elements such as network nodes, proxy nodes, and application nodes interconnected with a high-performance System Area Network fabric. The Web Accelerator project [14] looks at improving the performance of web server clusters by employing specialized nodes for HTTP processing and data delivery.²

The advantage of our half-pipe anchoring approach lies in the flexibility it affords for building heterogeneous systems. Since the state of the connection is maintained at the designated server, an optimal server need not even be running a TCP stack. The optimal server only has to be able to send packets, retransmit

²However, both approaches use backend forwarding to bring responses from where they are generated to the connection end point within the cluster.

previously sent packets, and garbage collect packets *as directed* by the connection handoff protocol. In particular, it need not be able to autonomously perform flow control and reliability operations.

IV. IMPLEMENTATION

A bidirectional TCP connection is actually made up of two unidirectional half-pipes. Our connection handoff protocol is based on the principle of anchoring the half-pipe from the client at one server node, while permitting the half-pipe to the client to move from one server node to another. We have implemented the connection handoff idea in the network stack, at the TCP level, of the Linux 2.4 kernel.

Placing the two half-pipes that make up a connection on different nodes forces us to introduce a facility that can tie them together to the extent required by TCP. In order to perform flow-control and reliability, TCP depends on the feedback it receives from one half-pipe when performing actions on the other half-pipe. Processing protocol information on the incoming half-pipe and affecting the state of the outgoing half-pipe are intimately tied together in TCP. When the two half-pipes need to be separated as dictated by our approach, we need to introduce a mechanism for communicating protocol information from one half-pipe to the other, and for affecting state changes.

We tie the two half-pipes together by introducing an *active* and a *passive* layer in the networking stack. The active layer executes on the designated server while the passive layer executes on the optimal server. The active layer is responsible for intercepting TCP actions that need to be performed on the outgoing half-pipe. It then relays these actions to the passive layer, which performs them on the outgoing half-pipe that exists at the optimal server. The active and passive layers communicate using the messages described below.

SETUP: The active layer sends this message to the passive layer in response to the execution of a handoff system call by the DS application:

```
handoff(socket, request, OS, OS-  
        port)
```

The current connection state of the socket is sent on the message to the passive layer, along with the data provided by the application to the handoff call. The active layer also marks the <OS, client, client-port, DS, DS-port> 5-tuple as a *GC-pending* half-pipe. On the opti-

mal server, the passive layer receives the SETUP message and creates a new socket using the provided connection state. The socket is created in the established state without the three-way TCP handshake with the client. The passive layer then provides the data sent on the message to the OS application waiting on OS-port. From this point onwards, any data sent by the application on the socket is handled through normal TCP actions.

SEND_PACKET and **GC_PACKET:** An incoming acknowledgement from the client is examined by the active layer. If it corresponds to data sent on an GC-pending half-pipe, the acknowledgement is split into the credit and garbage collection (GC) functionalities described in III-B.3. The location where the GC directive must be sent is determined by consulting the range of sequence numbers for each GC-pending half-pipe. If no corresponding half-pipe is found, the acknowledgement must be meant for the local node. The active layer also examines if the half-pipe for which the GC action is performed can be removed from the GC-pending list. This can be done if all packets sent by a half-pipe in the GC-pending list have been acknowledged, and the half-pipe has sent back DATA_DONE messages for all SETUP messages sent to it.

The location which can be given the credit to send a new packet will be the GC-pending half-pipe for which a DATA_DONE message has not yet been received. If no such half-pipe is found, the credit must belong to the local node. We then send the SEND_PACKET and GC_PACKET to the locations determined using the above rules. If both messages need to be sent to the same location, we piggyback one atop the other. If a credit and/or a GC directive is meant for the local node, we simply pass the incoming acknowledgement up the local protocol stack.

The passive layer converts both GC_PACKET and SEND_PACKET messages into vanilla acknowledgements and injects them up the protocol stack. The TCP stack on the optimal server views the acknowledgement as if it were sent from the client. It then performs the appropriate garbage collection actions and if necessary, sends a new packet to the client.

Note that a GC_PACKET converted by the passive layer back into an acknowledgement will not be wrongly interpreted by the passive-side protocol stack as an credit to send a new packet. This is because of

the rules governing the creation of the GC_PACKET on the designated server.

DATA_DONE: When the OS application closes the socket on which it is sending data to the client, the passive layer sends a DATA_DONE message to the active layer. This message contains the next sequence number that can be used for the connection. Upon receiving the DATA_DONE message, the active layer updates the connection state. It is only at this point that the active layer knows how many packets have been sent to the client on the half-pipe originating from the OS.

RESET: The passive layer needs to be informed when it can garbage collect the socket it created on the receipt of a SETUP message. Socket garbage collection can be done when all data has been sent and no unacknowledged packets remain. The active layer sends the RESET message when it removes a half-pipe from the GC-pending list.

The RESET message is also used to perform cleanup actions when the client-cluster half-pipe socket is garbage collected by the normal TCP protocol operations.

Since a RESET message garbage collects both the socket and any packets queued on it, a GC_PACKET message is subsumed by the RESET message. We use this property to coalesce acknowledgements for the last window of packets sent from an optimal server. By not sending GC_PACKET messages for the last window of packets, we can send a single garbage collection message for both the socket and the packets queued on it. This reduces the overhead of our protocol at the expense of increasing in-memory occupancy of the as-yet unacknowledged packets on the passive side.

KEEPALIVE: The active layer could send a KEEPALIVE message to all the passive layers that control GC-pending half-pipes at regular intervals. This enables a passive layer to know that the active layer on the designated server is functioning. In the absence of this message, the passive layer will time out and garbage collect the socket it is controlling. We have not yet implemented the KEEPALIVE message.

Our implementation currently requires that both nodes on which the two half-pipes reside have the TCP protocol stack. This requirement can be relaxed in a different implementation; in fact, an intriguing aspect

of our protocol is that the half-pipe to the client can originate from a node that does not have a TCP stack at all. Furthermore, while we have implemented the split-stack protocol over IP, it could just as well be implemented over a System Area Network such as the Virtual Interface Architecture [6].

V. EXPERIMENTAL EVALUATION

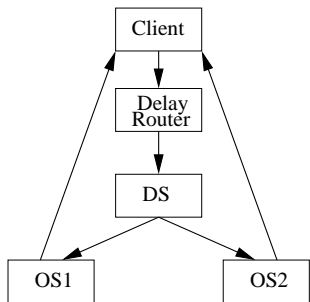
In this section, we report on our experience with our implementation and performance results obtained with our implementation. We have designed two experiments to illustrate the performance benefits of our connection handoff approach.

A. Experimental Environment

Our experimental environment consists of a set of 400 MHz Pentium II PCs interconnected using a switched 100 Mbps ethernet. Each node has 256 MBytes of memory and one Intel EtherExpress Pro/100 network interface. All nodes run the Linux 2.4.0-test4 kernel with our modifications to the TCP stack.

The web server we use in our experiments is Apache 1.3.19. Since our experiments compare two approaches, we are interested only in the relative performance of different connection handoff approaches on identical systems. Specifically, we do not consider the absolute performance of Apache. In experiments involving our half-pipe anchoring approach, we use a modified Apache server on the designated server. Our modification causes Apache to invoke the handoff system call when it determines that a request must be serviced at a different server node.

We used a modified version of the httperf workload generator [15]: httperf to drive our experiments. Our modifications enable httperf to maintain a sustained number of outstanding connections against a server. All of the results we report were obtained by maintaining a maximum of 150 outstanding connections on the server. We used a set of synthetic request streams as well as a partial trace of the web logs collected during the 1998 Winter Olympics in Nagano [16]. The average response size in the trace was 6KB with an average of 12 requests/connection.



DS: Designated Server
OS: Optimal Server

Fig. 4. Experimental setup for evaluating the effects of data pipe draining

B. Effect of Preventing Data Pipe Draining

In order to illustrate the benefits provided by connection anchoring, we compare the split-stack approach against one in which both half-pipes of the connection move from node to node. Anchoring the half-pipe from the client enables us to prevent the cluster-client data pipe from draining. In contrast, a traditional approach to connection handoff that moves the entire connection state forces the data pipe to drain between moves. We evaluate the split-stack approach against the traditional implementation using two metrics: the improvements in server throughput and the improvements in average connection lifetimes. The throughput measures the capacity of the cluster under constant load. The connection lifetime is the response time perceived by the client for one entire connection [17].

In a LAN setting, we expect the effect of pipe draining to be small. This is because both acknowledgements and data packets take very little time (a few milliseconds at most) to travel between the client and a cluster. In contrast, when a client connects to the cluster over a wide area network (WAN) packet delays in the hundreds of milliseconds can cause significant pipe draining effects.

Figure 4 illustrates the setup we used to measure the impact of pipe draining under real-world situations. We use a client running `httperf` to drive requests against a cluster composed of one designated server and two optimal servers. The designated server’s role is to transfer incoming requests on a client connection

to the OS best suited to service it. Response data is sent directly back to the client by the optimal server servicing the request. We consider two cases: the WAN case, and the LAN case. In the WAN case, we use the NIST Net delay router [18] to simulate WAN packet delays and drops. We use a delay of 175 ms, and a packet drop rate of 4% – these correspond to the global response time and packet drop rate reported by the Internet Traffic Report at the time this paper was written [19]. In the LAN case, we remove the delay router from the picture so that the client talks directly to the designated server.

In all the synthetic experiments, the designated server alternates the client requests on a connection between the optimal servers. The synthetic experiments depict a worst case scenario where every request in a connection “jumps” to (i.e. is serviced at) a different optimal server. The information from these experiments can be used to ascertain the benefits of the split-stack approach given a stream of connections that have a specific number of optimal server jumps per connection. This is because pipe draining effects are manifested *only* when a connection jumps from one optimal server to another.

In the case of the Olympics workload, the designated server uses the LARD locality based algorithm [1] to determine which optimal server should service a request.

Figures 5(a) and 5(b) illustrate the benefits of the split-stack approach as the number of jumps (requests) per connection is varied. Each request in this experiment retrieves a 6KB file along with a 230-byte HTTP header. Each figure considers two cases: a LAN case, and the WAN case. The client-side receive window is maintained at 16KB for this experiment. With an RTT of 175ms, this corresponds to a client-cluster bandwidth of 728Kb/s. As the figures show, the improvements in throughput of the split-stack approach increase almost linearly with the number of jumps. The improvements in average connection lifetimes (i.e., reduction in connection lifetimes) also scale linearly. These results are to be expected because in the traditional approach, each jump causes the data pipe to drain. Even at two jumps per connection, the split-stack approach gives a 40% performance improvement in throughput and a 27% improvement in connection lifetime. The absolute value in connection lifetime im-

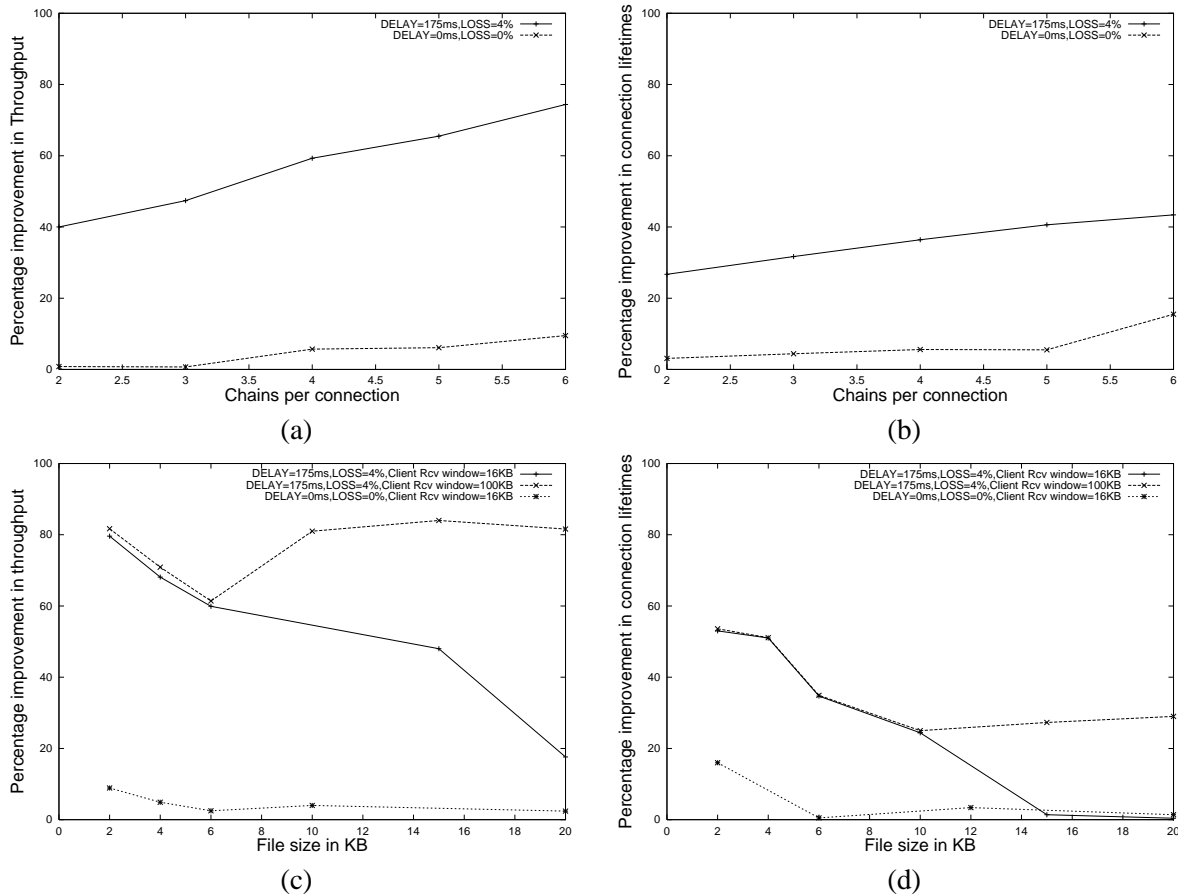


Fig. 5. Effects of pipe draining with varying jumps per connection

provements are fairly significant, with the lifetime improving from 1440 ms to 1055 ms at two jumps per connection, and improving from 3489 ms to 1972 ms at 6 req/conn.

Figures 5(c) and 5(d) illustrate the benefits of the split-stack approach as the file size is varied. The client sends four requests per connection throughout this experiment with each request being served on alternate optimal servers. Each graph contains three lines, corresponding to the LAN case, the global WAN case with a 16KB client receive window, and a global WAN case with a 100KB client receive window. The latter two cases correspond client-cluster bandwidths of 728Kb/s and 4.5Mb/s, as might be experienced by different classes of Internet users today and in the near future. As the figures show, with a 16KB client receive window, the improvement of the split-stack dwindles rapidly after a 10KB response size. However, with a larger client receive window, the improvements persist. With the 100KB client receive window, the im-

TABLE I
DATA PIPE DRAINING EFFECTS ON THE OLYMPICS98
TRACE

Expt. Setup	req/sec	Conn. lifetimes (ms)
LAN SS	1014	1140
LAN Trad. CH	903	1358
WAN SS	295	3356
WAN Trad. CH	244	4969

provements in connection lifetimes dip up to the 6KB point because of the way the TCP congestion window opens up. The improvements increase at 10KB and beyond because the split-stack prevents the data pipe from draining.

Table I shows similar results for the Olympics98 workload with a 100KB client receive window. The connection lifetimes show an improvement of 16% in the LAN case and 33% in the WAN case. The through-

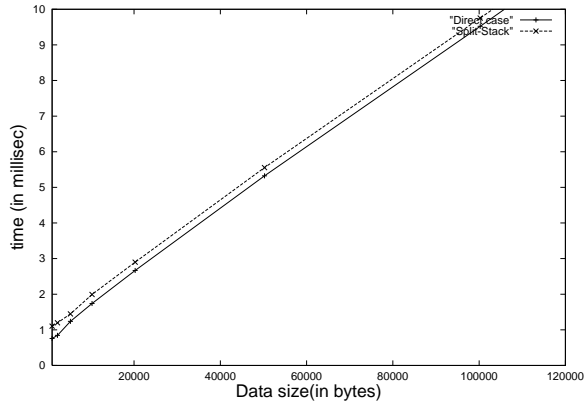


Fig. 6. Split-stack Protocol Overhead

put improvement was 12% in the LAN case and 21% in the WAN case.

C. Effect of Support for Heterogeneity

Figure 6 exposes the overhead of the split stack protocol by measuring the response time for one request on a connection. There are two cases in the graph. The solid line corresponds to the case where the client obtains a response directly from a server. The dashed line corresponds to a case where the client sends the request on a connection to one node (the DS), which then directs another node (the OS) to service the request and send the response. The difference between the two lines shows the pure overhead of the split-stack protocol as the file size is varied. As the file size increases, the overhead remains constant. This is expected because the split stack overhead is caused by the initial setup cost. Communication between the two half-pipes does *not* contribute any overhead because it is pipelined with the data transfer.

We have built a heterogenous system in which HTTP processing is done at the designated server, while data serving is relegated to the optimal server. An explanation of our system is in Section III-C.2. This architecture has been proposed by other researchers [14], [13]. However, we believe we are the first to implement such an approach where the generated response is sent directly back to the client from where it is generated *without* going through the client connection endpoint in the cluster.

VI. RELATED WORK

Limited mechanisms for connection handoff have been discussed in the context of web server clusters. Pai et al [1] discuss how to implement a single connection handoff between the front-end and the back-end designated server in a two-tier server cluster architecture. Their discussion does not cover connection handoff for persistent connections nor multiple connection handoff. Some of the same authors take on persistent connections in a later paper [7]. They provide an interesting analysis of the relative performance of connection handoff and back-end forwarding. Our paper has a different emphasis than theirs. Their analysis is carried on through simulations and their emphasis is on showing that content based request distribution policies can be very effective for persistent HTTP. Our emphasis is on understanding the benefits of a novel implementation of connection handoff that allows fine-grained load balancing of request coming over persistent HTTP connections independent of the load balancing policy being used.

Snoeren et al.[20] proposed end-to-end TCP connection migration protocol which provides server load balancing in the presence of long running connections. Since the protocol requires an end-to-end migration, its overhead is too large when compared to the service time of an individual request, making this strategy not desirable for short web transfers.

Shah et al.[13] propose a division of connection processing and application processing on different nodes in the cluster. Our implementation can support the heterogenous architecture that they advocate.

Bakre and Badrinath present Indirect-TCP to improve the throughput of TCP connections between highly mobile hosts and fixed hosts [8]. Connections between mobile hosts and fixed hosts are maintained using the help of the Mobile Support Router (MSR) of a cell, with the MSR indirectly opening connections to fixed hosts on behalf of the mobile host in its cell. When a mobile host switches cells, its TCP connections are handed off from the old to the new MSR. Handoff involves moving TCP connection state from one MSR to the other. Mobile IP [21] is then used to route all IP packets bound for the mobile host from the old MSR to the new MSR. I-TCP effectively hands off both half pipes that make up the connection. Consequently, two problems arise when it is applied in the

context of (immobile) server clusters. First, the data pipe from the MSRs (i.e., servers) will drain. Second, the cluster nodes to which connections can be handed off must all be (equally) capable of interpreting client requests. Our implementation of connection handoff addresses both problems

Haahr et. al. use a tunneling approach in “ALICE” to enable the use of CORBA in a mobile environment [11]. In ALICE, a mobile server informs its mobility gateway when it is willing to accept connections from clients. When the mobile server connects to a new mobility gateway (e.g. by switching cells), new logical connections corresponding to each existing connection are established from the new to the old gateway. ALICE thus tunnels connections at the session level from the old to the new gateway. While in ALICE connections handed off through multiple mobility gateways can result in a chain of mobility gateways involved in tunneling the connection, chaining is impossible in our approach. Also, ALICE’s handoff scheme is still in the design stage and has not been implemented.

VII. CONCLUSIONS

This paper presents Split–Stack, the first implementation of a connection handoff mechanism that can be efficiently applied to fine grained load balancing among the servers of a cluster. The key insight behind Split–Stack is to decouple the two unidirectional half-pipes that make up a TCP connection. Split–Stack anchors the unidirectional halfpipe 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 the server best suited to service the request. We have shown that split stack is very successful at addressing technical challenges, such as preventing the data pipe from draining during a handoff. We believe that one of the most interesting feature of Split–Stack is that it enable to explore the possibility to build a new breed of scalable server cluster, that exploit Split–Stack’s ability to support heterogeneous components to deliver high performance combined with high spatial density and low power consumption. We leave this exploration as future work.

REFERENCES

[1] 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 Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA.*, Oct. 1998.

[2] RFC 1945, “Hypertext transfer protocol – HTTP/1.0,” .

[3] J. Mogul, “The case for persistent-connection HTTP,” *In the Proceedings of the ACM SIGCOMM ’95 Symposium*, 1995.

[4] V. Padmanabhan and J. Mogul, “Improving HTTP latency,” *In the Proceedings of the Second International WWW Conference, Chicago, IL*, Oct 1994.

[5] RFC 2068, “Hypertext transfer protocol – HTTP/1.1,” .

[6] Virtual Interface Architecture Specification, ,” <http://www.viarch.org>.

[7] M. Aron, P. Druschel, and W. Zwaenepoel, “Efficient support for P-HTTP in cluster-based web servers,” *In Proceedings of the Usenix 1999 Annual Technical Conference, Monterey, CA.*, June 1999.

[8] A. Bakre and B. R. Badrinath, “Handoff and system support for indirect tcp/ip,” in *Proc. of MOBICOM95*, 1995.

[9] FL Omnichannel Technologies Inc., Boca Raton, ,” <http://www.omnichannel.com/Omni2FINAL.pdf>.

[10] TX RLX Technologies Inc., The Woodlands, ,” http://www.rlxtechnologies.com/product/features/server_blade.html.

[11] M. Haahr, R. Cunningham, and V. Cahill, “Supporting corba application in a mobile environment,” in *Proc. of MOBICOM99*, 1999.

[12] W. Richard Stevens, *TCP/IP Illustrated*, Addison–Wesley, 1994.

[13] Hemal V. Shah, Dave B. Minturn, Annie Foong, Gary L. McAlpine, Rajech S. Madukkarumukumana, and Greg J. Regnier, “Csp: A novel system architecture for scalable internet and communication services,” in *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.

[14] J. Song, E. Levy-Abegnoli, A. Iyengar, and D. Dias, “Design alternatives for scalable web server accelerators,” *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2000.

[15] D. Mosberger and T. Jin, “httperf – a tool for measuring web server performance,” in *in Proceedings of the SIGMETRICS Workshop on Internet Server Performance.*, 1998.

[16] 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.

[17] R. Rajamony and E. Elnozahy, “Measuring client perceived response times on the WWW,” in *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.

[18] NIST Net, ,” <http://snad.ncsl.nist.gov/itg/nistnet/>.

[19] Internet Traffic Report, ,” <http://www.internettrafficreport.com>.

[20] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan, “Fine-grained failover using connection migration,” in *Proc. of 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.

[21] C. Perkins, ,” RFC 2002, 1996.