Customized Library of Modules for STREAMS-based TCP/IP Implementation to Support Content-Aware Request Processing for Web Applications

Wenting Tang, Ludmila Cherkasova, Lance Russell
Hewlett-Packard Labs
1501 Page Mill Road
Palo Alto, CA 94303, USA
wenting,cherkasova,lrussell@hpl.hp.com

Matt W. Mutka
Dept of Computer Science & Eng.
Michigan State University
East Lansing, MI 48824, USA
mutka@cse.msu.edu

Abstract. *Content-aware request processing enables the intelligent routing and request processing inside the cluster to provide the quality of service requirements for different types of content and to improve overall cluster performance. STREAMS-based TCP/IP implementation in HP-UX 11.0 provides a convenient framework to design a library of new STREAMS modules to support content-aware request distribution and differentiation inside a cluster. Proposed modules take into account specifics of different cluster architectures and workload characteristics. These modules are designed as dynamically loadable modules and no changes are made to the existing TCP/IP code. The proposed design has the following advantages:* flexibility - *new modules may be loaded and unloaded dynamically, without node function interruption;* modularity - *proposed modules may be ported to other OSes with minimal effort. More importantly, the proposed STREAMS modules can be easily integrated and deployed into commercial OS systems, so the end users may take advantage of these solutions much sooner.*

1 Introduction

The replicated web server cluster is the most popular configuration used to meet the growing traffic demands imposed by the World Wide Web. However, for clusters to be able to achieve scalable performance as the cluster size increases, it is important to employ the mechanisms and policies for a balanced request distribution. As web sites become the platform to conduct the business, it is important to protect the web server from overload and to provide service differentiation when different client requests compete for limited server resources. Mechanisms for intelligent request distribution and request differentiation help to achieve scalable and predictable cluster performance and functionality, which are essential for today's Internet web sites.

Traditional request distribution try to distribute the requests among the nodes in the cluster based on parameters, such as IP addresses and port numbers, and some load information. Since the request distribution has the ability to check the packet header up to Layer 4 in OSI network reference model (in this case, TCP) when it makes the distribution decision. This is commonly referred as Layer 4 request distribution.

Content-aware request distribution takes into account the content (URL name, URL type, or cookies,etc) when making a decision to which server the request has to be routed. Content-aware request distribution mechanisms enable smart, specially tailored routing inside the cluster and provide many benefits. Some of the benefits are: 1) it allows the content of a web site to be only partially replicated. Dedicated nodes can be set up to deliver different types of documents. 2) it provides support for differentiated Web Quality of Service (WebQoS). 3) it can significantly improve the cluster throughput. Previous work on content-aware request distribution [6, 7, 1, 4] has shown that policies distributing the requests based on cache affinity lead to significant performance improvements compared to the strategies taking into account only load information.

Comparing to traditional Layer 4 request distribution, the complexity of content-aware request distribution lies in the fact that HTTP is a connection-oriented TCP protocol. In order to serve the client request (URL), a TCP connection has to be established between a client and a server node first. If the node cannot or should not serve the request, some mechanism has to be introduced to forward the request for processing to a right node in the cluster. *TCP splicing* and *TCP handoff* are two mechanisms proposed to support content-aware request distribution.

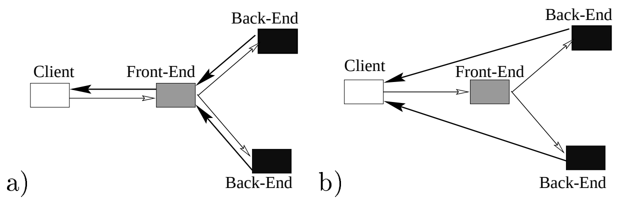

Figure 1: Traffic flow with a) TCP splicing mechanism; b) TCP handoff mechanism.

TCP splicing [5] is an optimization of the front-end relaying approach, with the traffic flow represented in Figure 1 a). In this cluster architecture, the front-end only dispatches the requests to the back-end node and it does not serve any requests at all. The TCP handoff mechanism was introduced in [6] to enable the forwarding of back-end responses directly to the clients without passing through the front-end, with traffic flow represented in Figure 1 b). After the front-end establishes the connection with the client, and the request distribution decision is made, the established connection is handed off to the selected back-end node to serve the request. The TCP state, related to the established connection, is migrated from the front-end to the selected back-end node. The main benefit of TCP handoff mechanism compared against TCP splicing is that the back-end node can send the response directly to the client. The front-end is not involved in the response data forwarding. It has been shown in [1] that TCP handoff mechanism provides better performance and scalability than TCP splicing.

STREAMS-based TCP/IP implementation, which is available in leading commercial operating systems, offers a framework to implement the TCP handoff mechanism as plug-in modules in the TCP/IP stack, and to achieve flexibility and portability without much of a performance penalty.

In this paper, we use three different applications to discuss specifics of content-aware request routing and related architectural design issues:

- a multi-language web site;
- partition-based cooperative web proxies;
- a simple e-commerce site.

Using these applications, we distinguish three most typical usage patterns of the TCP handoff mechanism. The usage pattern is defined by the fraction of the requests being handed off:

- *rare*-TCP handoff – when only a small fraction [1] of the requests are handed off for processing to a different cluster node;
- *frequent*-TCP handoff – when most of the requests are forwarded for processing to a different node using the TCP handoff mechanism;
- *always*-TCP handoff – when the requests are always handed off for processing to a different cluster node.

This difference in the usage patterns leads to different trade-off decisions in the modular implementation of TCP handoff mechanism. We discuss these trade-offs and propose a library of STREAMS modules implementing the TCP handoff functionality which addresses different cluster architectures and optimizes the TCP handoff mechanism for specific usage patterns.

Additionally, we discuss how content-aware request processing (CARP) can provide the necessary request differentiation and performance isolation, which are essential for today's business web site. The requests to the web site are classified into different classes. Request differentiation means that requests from different classes are assigned different priorities and high-priority traffic get preferred treatment in terms of resources. Performance isolation means that the requests are dispatched and scheduled in such a way that the certain throughput levels for a particular class are maintained relatively independent of the traffic from other classes. Kernel-level support of content-aware request processing, using a set of STREAMS modules, provides an interesting framework to implement differentiated services in a web server such as request classification, session management, request queuing, admission control, and/or request scheduling.

The proposed approach and a library of STREAMS modules (called **CARISMA**: **C**ontent-**A**ware, **R**equest **I**ntelligent **S**treams **M**odules libr**A**ry) have the following advantages:

- *portability*: the new modules are relatively independent of the implementation internals. New STREAMS modules are designed to satisfy the following requirements: all the interactions between new modules and the original TCP/IP modules are message-based, no direct function calls are made; new modules do not change any data structures or field values maintained by the original TCP/IP modules. This enables maximum portability, so that the new modules may be ported to other STREAMS-based TCP/IP implementation very quickly.
- *flexibility*: the new modules may be dynamically loaded and unloaded as DLKM (Dynamically Loadable Kernel Module) modules without service interruption.
- *transparency*: no application modification is necessary to take advantage of the new solutions. This is a valuable feature for applications where no source code is available.
- *efficiency*: the new modules are only peeking into the messages, with minimum functionality replication of the original TCP/IP modules.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to STREAMS and STREAMS-based TCP/IP implementations. Section 3 outlines three different web applications using content-aware request distribution and discusses the correspondent supporting architecture. Sections 4, 5, 6, argue that the efficient TCP handoff implementation in STREAMS environment should take into account the

[1] Small fraction means less than 50% of requests. However, in the applications we had considered, this portion can be only 5-20% of requests.

TCP handoff usage patterns to minimize the overhead introduced by the TCP handoff mechanism. The design of mechanisms to support content-aware request differentiation and processing is presented in Section 7.

2 STREAMS-Based TCP/IP Implementation

STREAMS is a modular framework for developing communication services. Each stream generally has a *stream head*, a *driver* and multiple optional *modules* between the stream head and the driver (see Figure 2 a). A stream is a full-duplex processing and data transfer path between a STREAMS driver and a process in user space. Modules exchange the information by *messages*. Messages can flow in two directions: *downstream* or *upstream*. Each module has a pair of *queues*: *write queue* and *read queue*. When a message passes through a queue, STREAMS modules for this queue are called to process the message. The modules may drop a message, pass a message, change the message header, and/or generate a new message.

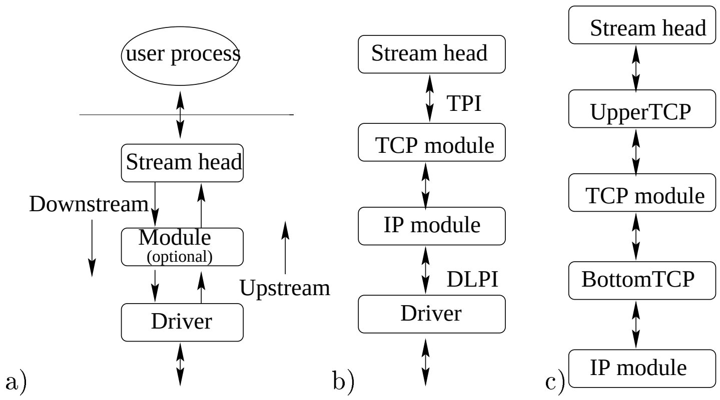

Figure 2: a) STREAMS b) STREAMS-Based TCP/IP Implementation c) New Plug-in Modules for rare-TCP Handoff in STREAMS-Based TCP/IP Implementation

The stream head is responsible for interacting with the user processes. It accepts the process request, translates it into appropriate messages, and sends the messages downstream. It is also responsible for signaling to the process when new data arrives or some unexpected event happens.

The STREAMS modules for STREAMS-based TCP/IP implementation are shown in Figure 2 b). Transport Provider Interface (TPI) specification [9] defines the message interface between TCP and upper module. Data Link Provider Interface (DLPI) specification [8] defines the message interface between driver and the IP module. These specifications define the message format, valid sequences of messages, and semantics of messages exchanged between the neighboring modules.

When the TCP module receives a SYN request for establishing the HTTP connection on the listen stream, the TCP module sends a T_CONN_IND message upstream. Under the TPI specification, TCP should not proceed until it gets response from the application layer. However, in order to be compatible with BSD implementation-based applications, the TCP module continues the connection establishment procedure with the client. When the application decides to accept the connection, it sends the T_CONN_RES downstream on the listen stream. It also creates another stream to accept this new connection, and TCP module attaches a TCP connection state to this new stream. The data exchange continues on the accepted stream until either end closes the connection.

3 Content-Aware Request Distribution: Cluster Design and Application Specific Issues

TCP handoff is the mechanism which enable the intelligent routing of web requests between cooperative nodes either locally or in the wide area. In this section, we use three different applications to discuss content-aware request routing and request processing and related architectural design issues.

In our discussion of different cluster designs which can be used to implement content-aware distribution strategies, we adopt the terminology proposed in [1]. There are three main components comprising a cluster configuration with content-aware request distribution strategy: *dispatcher* which implements the request distribution strategy, it decides which web server will be processing a given request; *distributor* which interfaces the client and implements the TCP handoff that distributes the client requests to specific web server; and *web server* which processes HTTP requests.

1. Multi-language web site design.

Big sites have different language versions to service different client community. For example, Yahoo has a site representation in different languages to serve different language groups. A client may access the same content in different languages. Another example is the big commercial companies which have on-line manual in different languages to serve different communities. Due to the volume of the traffic to these big sites, generally these sites are replicated at different places. It is not economical and necessary to replicate all different language versions of the same document in all the places at the same time. Typical practice is that servers at a particular place will only partially replicate a group of languages commonly used in the local community. It is desirable that a client is automatically directed to the right server to get the desired language document.

One simple and typical solution is to put a link on each page pointing to different versions of the same document in different languages, and let the user to select the right version manually. However, this method is not convenient for web page developers and very hard to manage in big sites.

An alternative solution is to apply the TCP handoff mechanism to automatically handoff the connection to the right server when the content is not present on the original server, and the selected server will respond to the client directly.

Each server in a cluster keeps a mapping (defined in a dispatcher module) to manipulate the URL according to some rules established in advanced by the site administrators. In particular, these rules assign the specific language versions to be served by different servers in a cluster. This is more flexible and convenient way to manage multi-language web site.

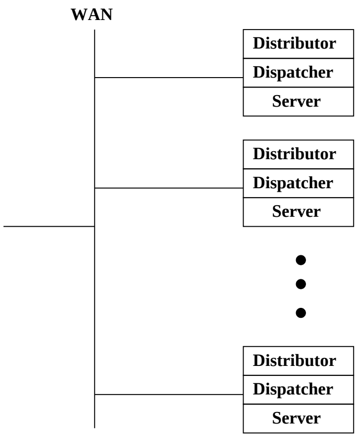

Figure 3: Web server cluster configurations with content-aware request distribution to support a multi-language web site design.

Figure 3 shows a cluster architecture to support a multi-language web site design. Web servers are connected by a wide area network, and thus, handoff has to be implemented over a wide area network. In this architecture, the distributor component is co-located with the server and dispatcher component.

For simplicity, we assume that the clients directly contact the distributor, for instance via Round-Robin DNS. In this case, the typical client request is processed in the following way. 1) Client web browser uses TCP/IP protocol to connect to the chosen distributor; 2) the distributor component accepts the connection and parses the request; 3) the distributor contacts the dispatcher for the assignment of the request to a server; 4) the distributor hands off the connection using TCP handoff protocol to the server chosen by the dispatcher; 5) the server takes over the connection using the handoff protocol; 6) the server application at the server node accepts the created connection; 7) the server sends the response directly to the client.

The specifics of this cluster architecture is that each node in a cluster has the same functionality: it combines a function of distributor and a web server. In other words, each node acts as a front-end and back-end node in providing TCP handoff functionality. For each web server, we expect that most of the HTTP requests are processed by the node accepting the connections (we refer to such requests as local), and hence TCP handoff happens infrequently. We use a term *rare-TCP handoff* to specify this usage pattern. Under such an usage pattern, a goal for the rare-TCP handoff design and implementation is a minimization of the overhead imposed by TCP handoff mechanism on local requests.

2. Partition-based cooperative web proxies design.

Web proxy is the typical place where intranet can access the Internet, it is very easy for a single proxy to become the bottleneck. In order to provide a scalable proxy service, cooperative proxy cluster is commonly used. One kind of cooperative proxy is partition-based proxies[10]. In partition-based cooperative proxies, each proxy caches a *disjoint* subset of the documents. Partition-based web proxy clusters increase the number of cached documents and improve the cache hit ratios.

However, the same partition function has to be applied by the browser to contact the correct proxy for a particular URL . Implementing the partition function in the browser-transparent way is a challenging and difficult task in partition-based proxy cluster.

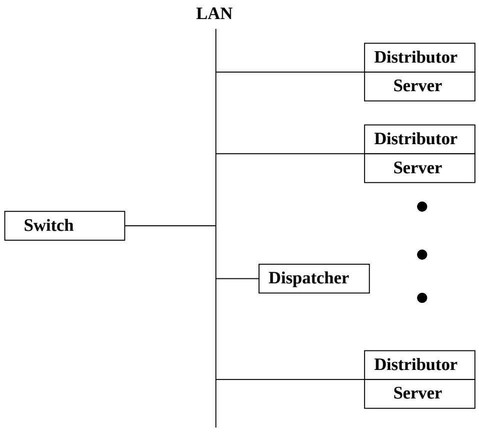

Figure 4: Partition-based cooperative web proxies design with content-aware request distribution.

TCP handoff can be used to implement the partition function in a client transparent manner. Figure 4 shows a cluster architecture to support a partition-based proxy cluster. In this architecture, the distributor component is co-located with the server component. The dispatcher component can be centralized as shown in Figure 4 or decentralized as shown in Figure 3 (typically, the decision is influenced by the choice of the partition function aiming to cache different files on a different servers). Round-Robin DNS or a front-end Layer 4 switch can be used to direct the traffic to the proxies in the cluster. When an HTTP request comes in, the proxy consults with a dispatcher module to determine which server should serve the request. If the request should be served by another proxy in the cluster, the original proxy hands the connection off to the designated proxy prescribed to process this request. The serving proxy will send the response back to the client directly.

This cluster architecture is similar to the architecture considered above for multi-language web site design. The difference is in the usage pattern of the TCP handoff. Let N be the number of nodes in the partition-based proxy cluster. Statistically, each node in the cluster will be serving only $1/N$ of the requests locally, while forwarding $(N-1)/N$ of the requests to the different nodes in the cluster using the TCP handoff mechanism. We use a term *frequent-TCP handoff* to specify this usage pattern. Under such an usage pattern, the efficient frequent-TCP handoff design and implementation should minimize the overhead from TCP handoff for remote request processing.

3. E-commerce site design.

HTTP protocol is stateless, i.e. each request is processed by the web server independently from the previous or subsequent requests. In e-commerce environment, a concept of *session* (i.e. a sequence of requests issued by the client) plays an essential role [2, 3]. For a session, it is important to maintain a state information from the previous interactions between a client and a server. Such state might contain the content of the shopping cart or list of results from the search request. Thus, when the user is ready to make a purchase, or is asking for the next 10 entries from the search list, the state information from the previous request should be retrieved. For efficient request processing and *session integrity*, it is desirable to send the client request to the same server. One of the popular scheme proposed for handling the state on the web is cookies. Content-aware request routing provides a convenient mechanism to support a session integrity (the other common term for this is "sticky" connection).

Figure 5 shows a cluster architecture to support a simple e-commerce site. In this architecture, the front-end node has co-located distributor and dispatcher modules to support the session integrity, i.e. based on the cookie attached to the URL, it sends the requests belonging to the same session to the initially chosen, same back-end server.

The specifics of this cluster architecture is that front-end and back-end nodes in a cluster have now different functionality: front-end combines a function of distributor and dispatcher, while the back-ends perform as web servers. The front-end node checks the cookie and decides which back-end server has to process the request. The distributor module always hands off the connection to the appropriate back-end server, front-end node never processes the request. We use a term *always-TCP handoff* to specify this usage pattern. Under such an usage pattern, the design and implementation of the always-TCP handoff is very different from the previously discussed cases of rare- and frequent-TCP handoff. The crucial difference is that front-end and back-end nodes play very different roles in this architecture.

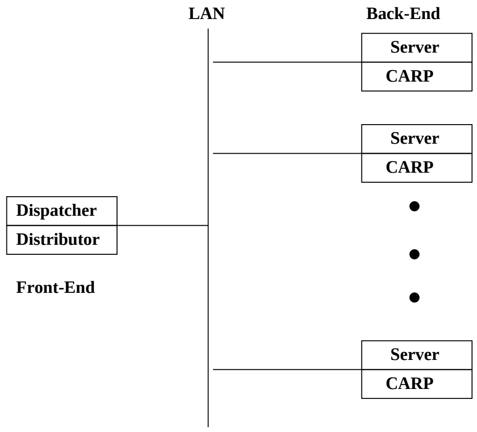

Figure 5: Web server cluster configurations to support session integrity and differentiated services for e-commerce site design.

Web server QoS is very important for business web sites. When a web site is overloaded, it is desirable that important requests get preferable service [2], or some form of the admission control mechanism is employed [3]. Content-aware requests processing (CARP) provides an interesting framework to implement differentiated services in a web server. For example, different request scheduling and processing could be deployed based on client or request priority. Session-based admission control, introduced in [3], can be easily implemented using CARP on the back-end web server nodes.

Thus, content-aware front-end node in the cluster configuration shown in Figure 5 provides the session integrity mechanism, while back-end web server nodes can deploy differentiated services such as request classification, session management, request queuing, admission control, and/or request scheduling.

This concludes our discussion on three different applications employing content-aware request processing and TCP handoff for different purposes. We illustrated the specifics of the TCP handoff usage patterns in these applications. Efficient implementation of the correspondent TCP handoff mechanism should take into account these usage patterns. The TCP handoff modules may be developed at different places in a TCP/IP stack to implement the content-aware request distribution, according to the architecture and workload characteristics.

Next Section 4 will present a detailed design of a rare-TCP handoff. Using this detailed description, we discuss what should be done differently for the efficient implementation of frequent-TCP handoff and always-TCP handoff. The design of mechanisms to support content-aware request differentiation and processing will be presented in Section 7.

4 Rare-TCP Handoff Design

In the cluster architecture shown in Figure 3, each node performs both front-end and back-end functionality: the distributor is co-located with the web server. We use the following denotations: the distributor-node accepting the original client connection request is referred to as FE (Front-End). In a case, when the request has to be processed by different node, this node receiving the TCP handoff request is referred to as BE (Back-End).

Two new modules are introduced to implement the functionality of rare-TCP handoff for multiple language web sites as shown in Figure 2 c). According to the relative position in the existing TCP/IP stack, we refer to the module right on top of the TCP module in the stack as UTCP (UpperTCP), and the module right under the TCP module as BTCP (BottomTCP).

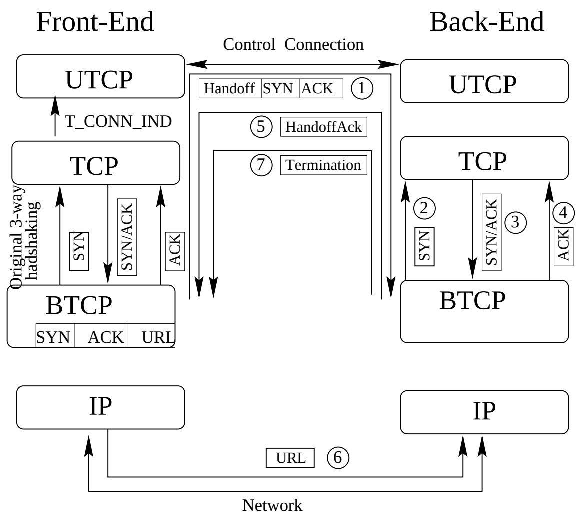

Figure 6: Remote Request Processing Flow During rare-TCP Handoff Procedure.

These two modules provide a wrapper around the current TCP module. In order to explain the proposed modular TCP handoff design and its implementation details, we consider a typical client request processing. There are two basic cases:
remote request processing, i.e. when the front-end node accepting the request must handoff the request to a different back-end node assigned to process this request;
local request processing, i.e. when the front-end node accepting the request is the node which is assigned to process this request.

First, we consider the *remote request* processing. There are six logical steps to perform the TCP handoff of the HTTP request in rare-TCP handoff:

1) finish 3-way TCP handshaking (connection establishment), and get the requested URL; 2) make the routing decision: which back-end node is assigned to process the request; 3) initiate the TCP handoff process with the assigned BE node; 4) migrate the TCP state from FE to BE node; 5) forward the data packets; 6) terminate the forwarding mode and release the related resources on FE after the connection is closed.

Now, we describe in detail how these steps are implemented by the newly added UTCP and BTCP modules and original TCP/IP modules in the operating system.

3-way TCP handshake. Before the requested URL is sent to make a routing decision, the connection has to be established between the client and the server. The proposed design depends on the original TCP/IP modules in the current operating system to finish the 3-way handshaking functionality. In this stage, $BTCP_{FE}$ allocates a connection structure corresponding to each connection request upon receiving a TCP SYN packet from the client. After that, $BTCP_{FE}$ sends the SYN packet upstream. Upon receiving a downstream TCP SYN/ACK packet from the TCP_{FE} module, $BTCP_{FE}$ records the initial sequence number associated with the connection, and sends the packet downstream. After $BTCP_{FE}$ receives an ACK packet from the client, it sends the packet upstream to TCP_{FE}. During this process, the $BTCP_{FE}$ emulates the TCP state transitions and changes its state accordingly.

In addition to monitoring the 3-way TCP handshaking, $BTCP_{FE}$ keeps a copy of the incoming packets for connection establishment (SYN packet, ACK to SYN/ACK packet sent by the client) and URL (Figure 6), for *TCP state migration* purpose, which is discussed later.

Also, because the TCP handoff should be transparent to server applications, the connection should not be exposed to the user level application before the routing decision is made. $UTCP_{FE}$ intercepts the T_CONN_IND message sent by TCP_{FE}. TCP_{FE} continues the 3-way handshaking without waiting for explicit messages from the modules on top of TCP.

URL parsing. $BTCP_{FE}$ parses the first data packet from the client, retrieves the URL and makes the distribution decision.

TCP handoff initiation. A special communication channel is needed to initiate the TCP handoff between FE and BE. A *Control Connection* is used for this purpose between two $UTCP_{FE}$ and $UTCP_{BE}$ as shown in Figure 6. This control connection is a pre-established persistent connection set up during the cluster initialization. Each node is connected to all other nodes in the cluster. The TCP handoff request is sent over the control connection to initiate the handoff process. Any communication between $BTCP_{FE}$ and $BTCP_{BE}$ modules goes through the control connection by sending the message to the $UTCP$ module first (see Figure 6). After $BTCP_{FE}$ decides to handoff the connection, it sends a handoff request to the $BTCP_{BE}$ (Figure 6, step 1). The SYN and ACK packets from the client and the TCP initial sequence number returned by TCP_{FE} are included in the message. $BTCP_{BE}$ uses the information in the handoff request to migrate the associated TCP state(steps 2-4 in Figure 6, which is discussed next). If $BTCP_{BE}$ successfully migrates the state, an acknowledgement is returned (Figure 6, step 5). $BTCP_{FE}$ frees the half-open TCP connection upon receiving the acknowledgement by sending a RST packet upstream to TCP_{FE} and enters forwarding mode. $UTCP_{FE}$ discards corresponding T_CONN_IND message when the T_DISCON_IND is received from the TCP_{FE}.

TCP state migration. It is not easy to get the current state of a connection at TCP_{FE}, to transfer it and to replicate this state at TCP_{BE}. First it is difficult to obtain the state out of the black box of the TCP module. Even if this could be done, it is difficult to replicate the state at BE. TPI does not support schemes by which a new half-open TCP connection with predefined state may be opened. In the proposed design, the half-open TCP connection is created by replaying the packets to the TCP_{BE} by the $BTCP_{BE}$. In this case, the $BTCP_{BE}$ acts as a client(Figure 6). $BTCP_{BE}$ uses the packets from $BTCP_{FE}$, updates the destination IP address of SYN packet to BE and sends it upstream (Figure 6, step 2). TCP_{BE} responds with SYN-ACK(Figure 6, step 3). $BTCP_{BE}$ records the initial sequence number of BE, discards SYN-ACK, updates the ACK packet header properly, and sends it upstream (Figure 6, step 4).

Packet forwarding. After the handoff is processed successfully, $BTCP_{FE}$ enters a forwarding mode. It forwards all the pending data in $BTCP_{FE}$, which includes the first data packet (containing the requested URL) (Figure 6, step 6). It continues to forward any packets on this connection until the forward session is closed.

During the packet forwarding step, $BTCP_{FE}$ updates (corrects) the following fields in the packet: 1) the destination IP address to BE's IP address; 2) the sequence number of the TCP packet; 3) the TCP checksum.

For packets that are sent directly from BE to the client, the $BTCP_{BE}$ module updates (corrects): 1) the source IP address to FE's IP address; 2) the sequence number; 3) TCP checksum. After that, $BTCP_{BE}$ sends the packet downstream.

Handoff connection termination. The connection termination should free states at BE and FE. The data structures at BE is closed by the STREAMS mechanism. $BTCP_{BE}$ monitors the status of the handoffed connection and notifies the $BTCP_{FE}$ upon the close of the handoffed connection in TCP_{BE} (Figure 6, step 7). $BTCP_{FE}$ releases the resources related to the forwarding mechanism after receiving such a notification.

Local request processing is performed in the following way. After the $BTCP_{FE}$ finds out that the request should be served locally, the $BTCP_{FE}$ notifies $UTCP_{FE}$ to release the correct T_CONN_IND message to upper STREAMS modules, and sends the data packet (containing the requested URL) to the original TCP module (TCP_{FE}). $BTCP_{FE}$ discards all the packets kept for this connection and frees the data structures associated with this connection. After this, $BTCP_{FE}$ and $UTCP_{FE}$ send packets upstream as quickly as possible without any extra processing overhead.

5 Frequent-TCP Handoff Design

Partition-based web proxy clusters demonstrate a different usage pattern of TCP handoff: HTTP requests are more likely to be handed off compared to rare-TCP handoff, as we pointed out in the section 3. This difference leads to a different TCP handoff design. In this design, the overhead of remote request should be minimized. The flow of the remote request processing is illustrated in Figure 7.

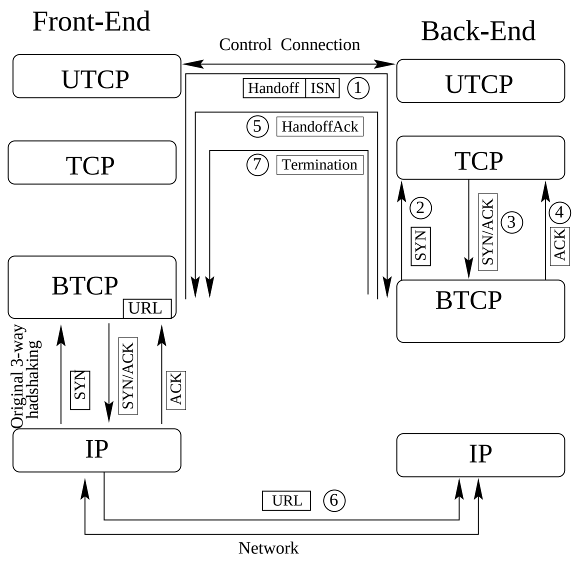

Figure 7: Remote Request Processing for Frequent-TCP Handoff

The additional modules are introduced at the same positions in the protocol stack as in the previous design, and are referred as B_FTCP and U_FTCP module, to indicate these modules have different functionalities.

Connection setup. Under rare-TCP handoff design, the connection-related resources in TCP module are released by RST message when the handoff is successful. In the frequent TCP handoff, it is inefficient to establish a TCP connection with the TCP module at the front-end node and then free the connection most of the time. Connection setup (the original 3-way handshaking) is reimplemented by the B_FTCP_{FE} module to trigger the client to send the URL. The B_FTCP_{FE} also has better control on TCP options. After B_FTCP_{FE} receives the URL and makes the decision, B_FTCP_{FE} may initiate the handoff connection through control connection as before (Figure 7, step 1). However, no packet is shipped along the persistent connection in the handoff request at this time. B_FTCP_{FE} may gather necessary information (for example, Initial Sequence Number (ISN), etc.) from the connection establishment packets, and then B_FTCP_{BE} may construct these packets from the information provided in the handoff request, and replay the packets locally at the back-end node (Figure 7, step 2-4). An acknowledgement is returned by the BE to the FE through the control connection to indicate that handoff is successful (Figure 7, step 5). If the request is processed locally at the front-end, the kept connection establishment packets are replayed to the local TCP module to create the necessary state.

Packet forwarding. Packets may be forwarded to the selected server on top of IP layer, in the IP layer, or under IP layer, depending on the cluster configuration and the ratio between the local traffic and forwarding traffic. While $B_F TCP$ module may forward the packets on top of IP layer, similar functionalities can be achieved by inserting a module on top of device driver.

When $B_F TCP$ is implemented on top of the device driver, and all the back-end nodes are located on the same LAN (as in the described partion-based proxy application), it is possible for a cluster to have a virtual IP address, each back-end node is uniquely identified by MAC address, and the packet is forwarded by filling in the right MAC address. This avoids the Network Address Translation (NAT) on the front-end and the NAT on the back-end node for outgoing traffic. Upon receiving the DLPI message from the device driver, $B_F TCP_{FE}$ changes the DLPI message format and destination MAC address and sends the message downstream.

When the forwarding packets may need to traverse a router or across a WAN, packet's destination may be changed to the selected server's IP address and special protocol may be developed to carry the packet's original IP address to the selected server so that the response packet's source IP address may be updated accordingly. $B_F TCP_{FE}$ updates the packet's IP address to the selected server's IP address, and sends the packet upstream. The IP_{FE} forwards the packet according to its routing tables to the back-end node. $B_F TCP_{BE}$ has to manipulate the TCP header anyway and updates the initial sequence number and TCP checksum.

The original TCP connection establishment can be done in two different modules: either in operating system TCP module or $B_F TCP$ module. When TCP module is used to establish the connection with the client, the initial sequence number is correct so local request may respond to the client directly without any packet header manipulation. If the $B_F TCP$ module is used, because of the local handoff, the initial sequence number used by the TCP module and BTCP module might be different so that $B_F TCP$ has to update the initial sequence number and TCP checksum for every outgoing packet for local requests. In this design, we improve the remote request processing at a price of an additional small penalty for the local requests processing.

Handoff connection termination. In order to successfully and robustly free the front-end forward session, either back-end or the front-end node has to observe the two-way TCP control traffic. In the rare-TCP handoff design, the $BTCP_{BE}$ sees the two-way traffic and knows the status of the TCP connection and sends the notification to the front-end upon the termination of the connection.

In frequent-TCP handoff design, connection termination depends on where the modules are inserted. If the modules is on top of IP level, the rare-TCP handoff termination approach is more elegant. If the functionality of the $B_F TCP$ is implemented by a module inserted on top of device driver, the termination approach described in the next section for always-TCP handoff is better[2].

6 Always-TCP Handoff Design

In always-TCP handoff, there are two kind of nodes, the dedicated front-end node and the back-end web servers. The purpose of the front-end node is to trigger the client to send the URL, and then handoff the connection to the selected server. The request flow of always-TCP handoff is shown in Figure 8. In this configuration, only one module is introduced, $B_A TCP$ module, at both the front-end and the back-end nodes. The difference be-

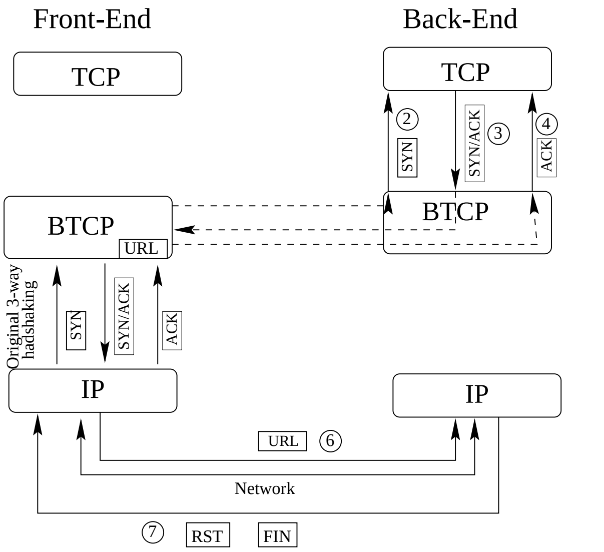

Figure 8: Request Processing for always-TCP Handoff

tween always-TCP handoff and the previously described rare- and frequent-TCP handoff is as follows.

Connection setup. The $B_A TCP_{FE}$ implements the connection setup function as in $B_F TCP_{FE}$ module. Packets are exchanged to establish the connection with the client and get the URL. The state migration is done by replaying the packets between the front-end and the back-end node. Since all web traffic is handoff traffic, a TCP SYN packet arrived at web server listening port indicates a handoff request is initiated (Figure 8, step 1) and $B_A TCP_{BE}$ sends the SYN packet upstream

[2]The module on top of the device driver may not see two-way traffic in case the multiple network interface cards exist, and incoming traffic and outgoing traffic go through different interfaces.

(Figure 8, step 2). TCP persistent connection may not be needed because there is no need to tell the back-end node that a particular flow is the handoff connection.

The SYN-ACK packet from TCP_{BE} is intercepted by B_ATCP_{BE} to the front-end by changing IP address (Figure 8, step 3). The B_ATCP_{BE} receives ACK and records the initial sequence number returned by the B_ATCP_{FE} (Figure 8, step 4). No acknowledgement is needed. URL is forwarded as before in step 6.

Packet forwarding should be done at quickly as possible. In this configuration, it might be better to forward the packet on top of the device driver, also virtual IP address should be used to avoid network address translation at the front-end because it is much easy for the front-end to become the bottleneck for the whole cluster.

Handoff connection termination. The handoff connection is closed in the following fashion. The B_ATCP_{BE} intercepts the TCP control packets (packets with flags on, for example, RST, FIN) and sends it to the B_ATCP_{FE} (step 7). The B_ATCP_{FE} records the connection progress and relays the packets to the client. Data traffic goes directly to the client. The front-end sees two way traffic and may keep track of handoff connection status and closes the connection in timely manner.

7 Content-Aware Request Processing and Differentiation

Content-aware request processing can provide the necessary request differentiation and performance isolation, which is essential to today's business web site. Request differentiation may be supported either at user level, or at kernel level by content-aware protocol stack. At user level, the request differentiation is typically implemented by the distributing process, which accepts all the incoming HTTP requests, classifies the requests into one of several classes (queues). The working processes get the requests from the queues, and process the requests. The distribution process may be implemented by the web server software itself, or may be supported by a separate process that feeds the requests to the web server software transparently.

Kernel based request differentiation has the following advantages:

No extra bottlenecks. User-level implementation introduces the distribution process, which controls all the incoming traffic, and classifies them into different priorities queues. That introduces another central control point, which might become a bottleneck. The number of distributing processes that are sufficient to process the incoming requests efficiently is highly workload dependent. Kernel-level implementation does not introduce the additional central control points except the ones existing in the kernel already.

Less resource consumption. When the server reaches the overloading point, the admission control has to take place. It is strongly desirable that if the server decides to reject the request (or a new session), a simple rejection message is returned so that the user will not try to submit the same request again and again. For such requests, processing them as quickly and efficiently as possible is critical. User-level implementation has to accept the connection, get the request, and return a message. Kernel-level implementation, performing similar actions, is much more efficient: it does not introduce context switches and socket buffers.

Efficient measurements. Kernel-level implementation may get accurate information quickly and accurately compared to user-level implementation. These measurements may be important to the decision process, such as the number of connections open at given moment, activities on each open connection, average response time, the network bandwidth and roundtrip times of each connection, etc. The kernel implementation may take advantages of these measurements and make more intelligent decision.

Content-aware request processing is designed to perform the following functions:

1. Request Classification. The classification identifies a request and assigns it to a class. Information within the request or provided by the application is applied against the classification policies to determine the request class.

The back-end node depends on the operating system TCP module to finish three-way handshaking. The UTCP in back-end node has the necessary number of priority queues and a partially-finished connection queue. BTCP creates a simple structure for each connection and sends the connection establishment packets upstream. UTCP holds the corresponding T_CONN_IND message into the partially-finished connection queue. After TCP module finishes three-way handshaking with client, the client sends the URL to the server. BTCP retrieves the URL after receiving the packet, and classifies the request according to the policy specified by the administrator. BTCP sends the classification of the URL to the UTCP, and UTCP places the corresponding T_CONN_IND message from the partially finished connection queue to one of the supported class queues.

This design may support the persistent connections. Since for persistent connections, the connection has been already established and exposed to the application, UTCP intercepts the subsequent requests, and classifies them into one of the classes, and places the message (T_DATA_IND) in one of the queues.

2. Admission Control. The admission control mechanism prevents server from overload. The classes of requests, the admission control policies and the system or service load measurements may be used to determine whether to accept a request (or a new session). Action policies may specify redirecting traffic to other nodes inside or outside of the cluster, or return a message indicating that the web server is currently busy.

The admission control mechanism can be deployed using BTCP module. First, BTCP checks the URL (or cookie), if this request can not be served at this time, the BTCP sends the message to the client and releases the connection by sending a RST message upstream.

For subsequent requests on a persistent connection, UTCP checks the URL (cookie) from the T_DATA_IND message, and makes the decision. If such a request can not be served at this time, UTCP sends the customized message to TCP module and releases the connection. The returned message may be a redirection to other servers, or a customized message stating that the server is busy at this moment.

3. Request Scheduling. Request scheduling is used to provide performance isolation or differentiation depending on the scheduling and classification policies. UTCP module may support a set of request scheduling strategies, for example, FIFO, fair-sharing, weighted fair-sharing, strict priority queue, etc.

Some advanced request scheduling policies might be supported too. Additionally, UTCP and BTCP may take some measurements, so the rate-based scheduling may be used to guarantee a minimum share of the throughput to a particular class.

8 Conclusion

Research on request distribution and request differentiation receives much attention from both industry and academia. Providing scalable and predictable service is essential for future Internet web sites. Content-aware request processing enables intelligent routing and request processing inside the web cluster to support the quality of service requirements for different types of content and to improve overall cluster performance.

STREAMS-based TCP/IP implementation, which is available in leading commercial operating systems, offers a framework to implement the TCP handoff mechanism as plug-in modules in the TCP/IP stack.

In this paper, we use three different applications to discuss specifics of content-aware request routing and related architectural design issues. The difference in the usage patterns leads to different trade-off decisions in the modular implementation of TCP handoff mechanism. We discuss these trade-offs and propose a library of STREAMS modules implementing the TCP handoff functionality which addresses different cluster architectures and optimizes the TCP handoff mechanism for specific usage patterns. Additionally we discuss kernel-level support of content-aware request processing, using a set of STREAMS modules. This proposes an interesting framework to implement differentiated services in a web server such as request classification, session management, request queuing, admission control, and/or request scheduling.

The library of STREAMS modules, proposed in this paper, offers a set of attractive benefits: *portability*, *flexibility*, *transparency*, and *efficiency* to support scalable web server cluster design and smart, specially tailored request routing inside the cluster. More importantly, these modules allow easier integration into commercial systems so that end user may benefit from them sooner.

References

[1] M. Aron, D. Sanders, P. Druschel, W. Zwaenepoel. Scalable Content-Aware Request Distribution in Cluster-based Network Servers. In Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, CA, June 2000.

[2] N. Bhatti, R. Friedrich. Web Server Support for Tiered Services. IEEE Network J., vol. 13, no. 5, Sept-Oct, 1999, pp. 64-71.

[3] L. Cherkasova, P. Phaal. Session Based Admission Control: a Mechanism for Improving Performance of Commercial Web Sites. In Proceedings of Seventh International Workshop on Quality of Service, IEEE/IFIP event, London, May 31-June 4, 1999.

[4] L. Cherkasova. FLEX: Load Balancing and Management Strategy for Scalable Web Hosting Service. In Proceedings of the Fifth International Symposium on Computers and Communications (ISCC'00), Antibes, France, July 3-7, 2000, p.8-13.

[5] A. Cohen, S. Rangarajan, and H. Slye. One the Performance of TCP Splicing for URL-Aware redirection. In Proceedings of the 2nd Usenix Symposium on Internet Technologies and Systems, Boulder, CO, Oct, 1999.

[6] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Drushel, W. Zwaenepoel, E. Nahum: Locality-Aware Request Distribution in Cluster-Based Network Servers. In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, 1998, pp. 205-216.

[7] X. Zhang, M. Barrientos, J. Chen, M. Seltzer: HACC: An Architecture for Cluster-Based Web Servers. In Proceeding of the 3rd USENIX Windows NT Symposium, Seattle, WA, July, 1999.

[8] Data Link Provider Interface (DLPI), UNIX International, OSI Work Group.

[9] Transport Provider Interface (TPI), UNIX International, OSI Work Group.

[10] D. Karger and A. Sherman and A. Berkheimer: Web Caching with Consistent Hashing, Proceedings of 8th World Wide Web Conference, May,1999.