Modular TCP Handoff Design in STREAMS-Based TCP/IP Implementation

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 Universsity
East Lansing, MI 48824, USA
mutka@cse.msu.edu

Abstract. Content-aware request distribution is a technique which takes into account the content of the request when distributing the requests in a web server cluster. A handoff protocol and TCP handoff mechanism were introduced to support content-aware request distribution in a client-transparent manner. Content-aware request distribution mechanisms enable the intelligent routing inside the cluster to provide the quality of service requirements for different types of content and to improve overall cluster performance.
We propose a new modular TCP handoff design based on STREAMS-based TCP/IP implementation in HP-UX 11.0. We design the handoff functions as dynamically loadable modules. No changes are made to the existing TCP/IP code. The proposed plug-in module approach has the following advantages: *flexibility*-TCP handoff functions may be loaded and unloaded dynamically, without node function interruption; *modularity*- proposed design and implementation may be ported to other OSes with minimal effort.

1 Introduction

The 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.

The market now offers several hardware/software load-balancer solutions that can distribute incoming stream of requests among a group of web servers. Typically, the load-balancer sits as a front-end node on network and acts as a gateway for incoming connections (we often will call this entity a distributor). Incoming client requests are distributed more or less evenly to a pool of servers (back-end nodes).

Traditional load balancing solutions for a web server cluster try to distribute the requests among the nodes in the cluster based on some load information without regard to the requested content and therefore forwarding the client requests to a back-end node prior to establishing a connection with the client.

Content-aware request distribution takes into account the content (such as URL name, URL type, or cookies) when making a decision to which server the request is to be routed. Previous work on content-aware request distribution [4, 5, 1, 2] 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.

HTTP protocol relies on TCP - a connection-oriented transport protocol. The front-end node (the request distributor) must establish a connection with the client to inspect the target content of a request prior to asssigning the connection to a back-end web server. A mechanism is needed to service the client request by the selected back-end node. Two methods were proposed to distribute and service the requests on the basis of the requested content in a client-transparent manner: the TCP handoff [4] and the TCP splicing. [3].

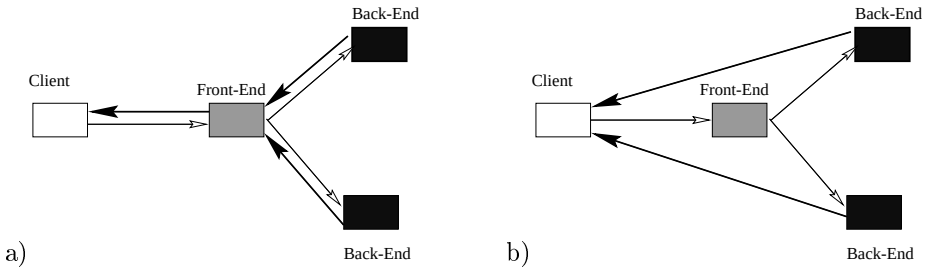

Fig. 1. Traffic flow with a) TCP splicing; b) TCP handoff.

TCP splicing is an optimization of the front-end relaying approach, with the traffic flow represented in Figure 1 a).

The TCP handoff mechanism was introduced in [4] 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). The main idea behind the TCP handoff is to migrate the created TCP state from the distributor to the back-end node. The TCP implementation running on the front-end and back-ends needs a small amount of additional support for handoff. In particular, the protocol module needs to support an operation that allows the TCP handoff protocol to create a TCP connection at the back-end without going through the TCP three-way handshake with the client. Similar, an operation is required that retrieves the state of an established connection and destroys the connection state without going through the normal message handshake required to close a TCP connection. Once the connection is handed off to a back-end node, the front-end must forward packets from the client to the appropriate back-end node.

This difference in the response flow route allows substantially higher scalability of the TCP handoff mechanism than TCP splicing. In [1], authors compared performance of both mechanisms showing the benefits of the TCP handoff schema. Their comparison is based on the implementation of the TCP handoff mechanism in FreeBSD UNIX.

In this work, we consider a web cluster in which the content-aware distribution is performed by each node in a web cluster. Each server in a cluster may forward a request to another node based on the requested content (using TCP handoff mechanism).

STREAMS-based TCP/IP implementations, which are 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 the flexibility and portability without too much performance penalty. As part of the effort to support a content-aware request distribution for web server clusters, we propose a new modular TCP handoff design. The proposed TCP handoff design is implemented as STREAMS modules. Such a design has the following

advantages:

- *portability*: the STREAMS-based TCP/IP modules are relatively independent of the implementation internals. New TCP handoff modules are designed to satisfy the following requirements:
  - all the interactions between TCP handoff modules and the original TCP/IP modules are message-based, no direct function calls are made.
  - TCP handoff modules do not access and/or change any data structures or field values maintained by the original TCP/IP modules.

  This enables maximum portability, so that the TCP handoff modules may be ported to other STREAMS-based TCP/IP implementation very quickly.
- *flexibility*: TCP handoff 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 TCP handoff mechanism. This is a valuable feature for some applications where no source code is available.
- *efficiency*: the proposed TCP handoff modules are only peeking into the messages, with minimal functionality replicated from the original TCP/IP modules.

2 Cluster Architectures and Request Distribution Mechanisms

Different products have been introduced in the market for load balancing.

Popular Round-Robin DNS solutions [8] distribute the accesses among the nodes in the cluster in the following way: for a name resolution it returns the IP address list (for example, list of nodes in a cluster which can serve this content), placing a different address first in the list for each successive request. Round-Robin DNS is available as part of DNS which is already in use, i.e. there is no additional cost.

Other traditional load balancing solutions for a web server cluster try to **distribute the requests** among the nodes in the cluster without regard to the requested content and therefore forwarding client requests to a back-end node **prior to establishing a connection** with the client as shown in Figure 2. In this configuration, web server cluster appears as a single host to the clients. To the back-end web servers, the front-end load-balancer appears as a gateway. In essence, it intercepts the incoming web requests and determines which web server should get each one. Making that decision is the job of the proprietary algorithms implemented in these products. This code can take into account the number of servers available, the resources (CPU speed and memory) of each, and how many active TCP sessions are being serviced, etc. The balancing methods across different load-balancing servers vary, but in general, the idea is to forward the request to the least loaded server in a cluster.

Only the virtual address is advertised to the Internet community, so the load balancer also acts as a safety net. The IP addresses of the individual servers are never sent back to the web browser. The load-balancer rewrites the virtual cluster IP address to a particular web server IP address using *Network Address Translation* (NAT). Because of this IP address rewriting, both inbound requests and outbound responses pass through the load-balancer.

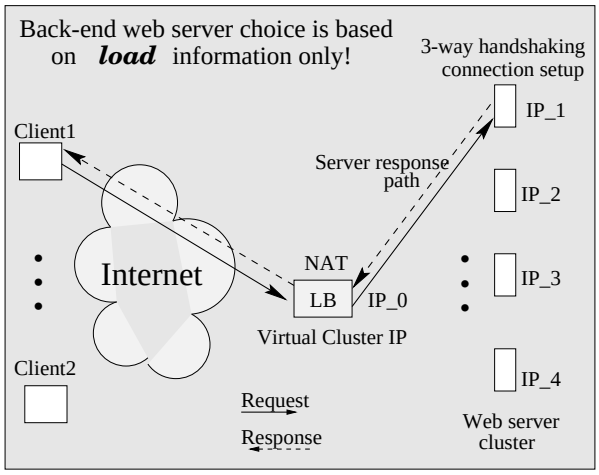

Fig. 2. Traditional load balancing solution (like Cisco's Local Director) in a web server cluster.

The **3-way handshaking and the connection set up** with original client is the responsibility of the chosen **back-end web server**. After the connection is established, the client sends to this server the HTTP request with specific URL to retrieve.

Content-aware request distribution intends to take into account the content (such as URL name, URL type, or cookies) when making a decision to which server the request has to be routed. The main technical difficulty of this approach is that it requires the establishment of a connection between the client and the request distributor. So the client will send the HTTP request to the distributor. The distributor can then make a decision to which back-end web server this request will be forwarded.

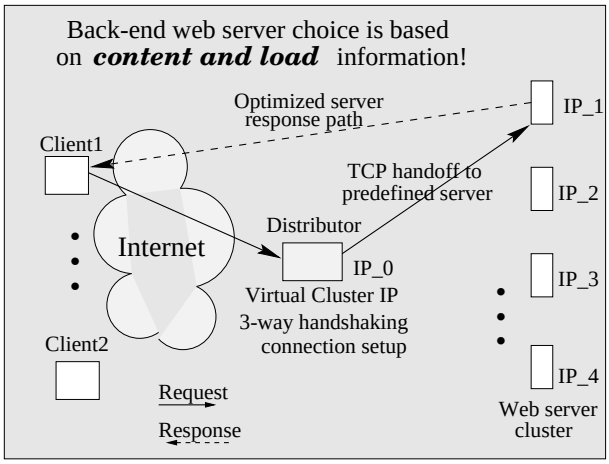

Fig. 3. Content-aware request distribution solution (front-end based configuration) in a web server cluster.

Thus, **3-way handshaking and the connection set up** between the client and **request distributor** happens first, as shown in Figure 3. After that, back-end web server is chosen based on the content of the HTTP request from the client. To be able to distribute the requests on the basis of requested content, the distributor component should implement either a form of TCP handoff [4] or the splicing mechanism [3]. Figure 3 shows request and response flow in case of TCP handoff mechanism.

In this configuration, the typical bottleneck is due to the front-end node which performs the functions of distributor. For realistic workloads, a front-end node, performing the TCP handoff, does not scale far beyond four cluster nodes [1]. Most of the overhead in this scenario is incurred by the distributor component.

Thus, another recent solution proposed in [1] is shown in Figure 4. It is based on alternative cluster design where the **distributor is co-located with the web server**. We will call this architecture CARD (Content-Aware Request Distribution).

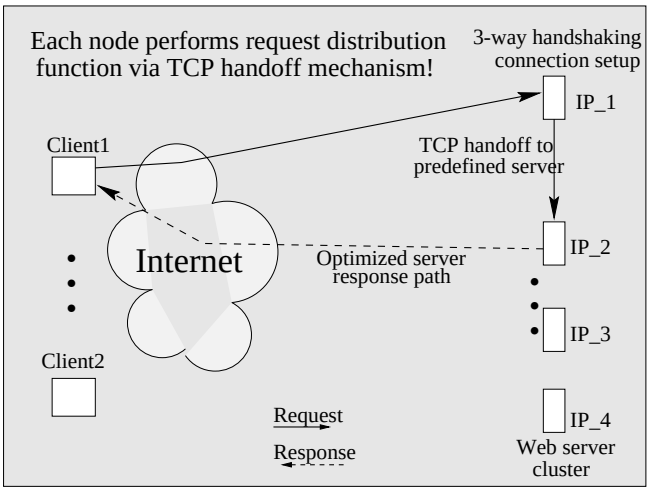

Fig. 4. Content-aware request distribution solution (cluster based, distributed configuration) in a web server cluster.

For simplicity, we assume that the clients directly contact the distributor, for instance via RR-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, and decides on server assignment for this request; 3) the distributor hands off the connection using TCP handoff protocol to the chosen server; 4) the server application at the server node accepts the created connection; 5) the server sends the response directly to the client.

The results in [1] show good scalability properties of the CARD architecture when distributing requests with the LARD policy [4]. The main idea behind LARD is to partition the documents logically among the cluster nodes, aiming to optimize the usage of the overall cluster RAM. Thus, the requests to the same document will be served by the same cluster node that will most likely have the file in RAM.

Our TCP handoff modules are designed to support a content-aware request distribution for the CARD implementation shown in Figure 4.

3 STREAMS and STREAMS-Based TCP/IP Implementation

STREAMS is a modular framework for developing the communication services. Each stream has a *stream head*, a *driver* and multiple optional *modules* between the stream head and the driver (see Figure 5 a). 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, the service routine for this queue may be called to process the message. The service routine may drop a message, pass a message, change the message header, and generate a new message.

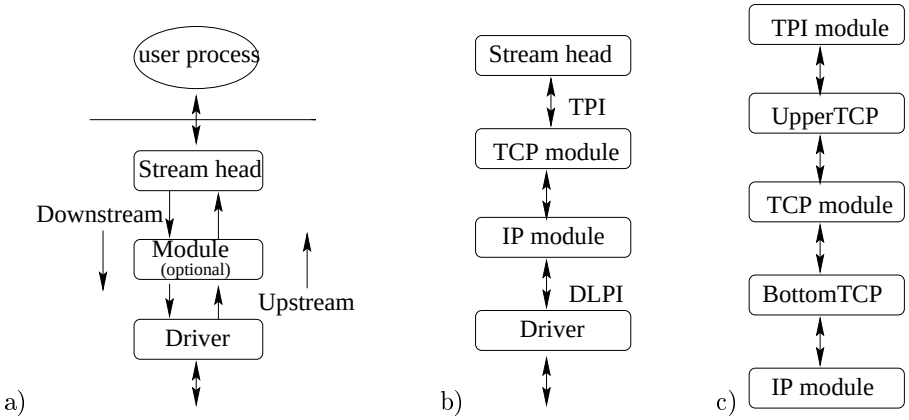

Fig. 5. a) STREAMS b) STREAMS-Based TCP/IP Implementation c) New Plug-in Modules for TCP Handoff in STREAMS-Based TCP/IP Implementation

The stream head is responsible for interacting with the user processes. It accepts the process request, translates it into appropriate messages, and sends 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 5 b). Transport Provider Interface (TPI) specification [7] defines the message interface between TCP and the upper module. Data Link Provider Interface (DLPI) specification [6] 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 these neighboring modules.

When the TCP module receives a SYN request for establishing the HTTP connection, the TCP module sends a T_CONN_IND message upstream. Under the TPI specification, TCP should not proceed until it gets the 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. It also creates another stream to accept this new connection, and TCP module attaches the TCP connection state to this new stream. The data exchange continues on the accepted stream until either end closes the connection.

4 Modular TCP Handoff Design

The TCP handoff mechanism (shown in Figure 1 b) enables the response forwarding from the back-end web server nodes directly to the clients without passing through the distributing front-end.

In the CARD architecture, 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 the case where the request has to be processed by a different node, thenode receiving the TCP handoff request is referred to as BE (Back-End).

Two new modules are introduced to implement the functionality of TCP handoff as shown in Figure 5 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).

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 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 the CARD architecture:

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.

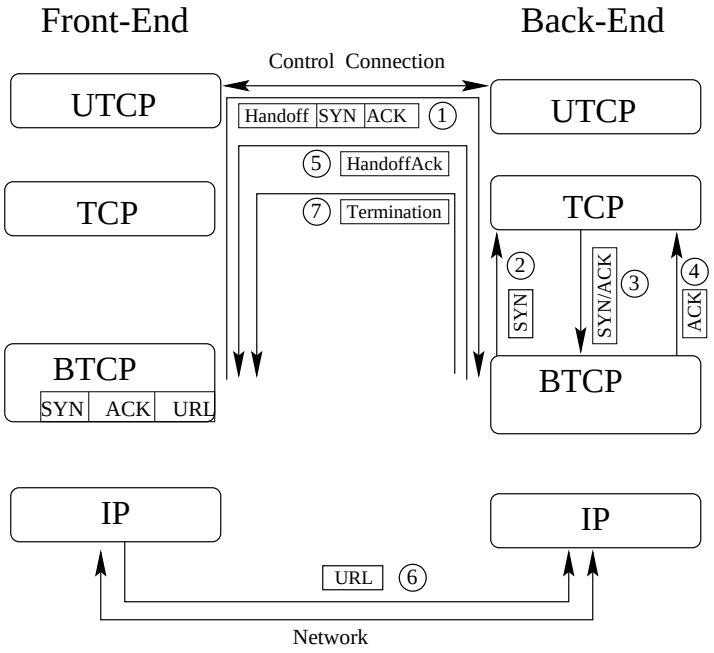

Fig. 6. Remote Request Processing Flow During TCP Handoff Procedure

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 are 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*
  In the STREAMS environment 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).
- *Data 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 data 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 data 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 Conclusion

Research on scalable web server clusters has received much attention from both industry and academia. A routing mechanism for distributing requests to individual servers in a cluster is at the heart of any server clustering technique. Content-aware request distribution (LARD, HACC, and FLEX strategies) [4, 5, 1, 2] has shown that policies distributing the requests based on cache affinity lead to significant performance improvements compared to the strategies taking into account only the load information.

Content-aware request distribution mechanisms enable intelligent routing inside the cluster to support additional quality of service requirements for different types of content and to improve overall cluster performance.

With content-aware distribution, based on TCP handoff mechanism, incoming requests must be handed off by distributor component to a back-end web server in a client-transparent way after the distributor has inspected the content of the request. The modular TCP handoff design proposed in this paper offers additional advantages: *portability*, *flexibility*, *transparency*, and *efficiency* to support scalable web server cluster design and smart request routing inside the cluster.

References

1. Mohit Aron, Darren Sanders, Peter Druschel and Willy 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. 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.
3. 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.
4. 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 (ASPLOS VIII), ACM SIGPLAN,1998, pp.205-216.
5. 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.
6. Data Link Provider Interface (DLPI), UNIX International, OSI Work Group.
7. Transport Provider Interface (TPI), UNIX International, OSI Work Group.
8. T. Brisco: DNS Support for Load Balancing. RFC 1794, Rutgers University, April 1995.