KNITS: Switch-based Connection Hand-off

Athanasios E. Papathanasiou
University of Rochester
Department of Computer Science
PO Box 270226
Rochester, NY 14627
Email: papathan@cs.rochester.edu

Eric Van Hensbergen
International Business Machines
Austin Research Lab
11400 Burnet Road, MS 9460
Austin, TX 78758
Email: bergevan@us.ibm.com

Abstract—**This paper describes a mechanism allowing nodes to hand-off active connections by utilizing connection splicing at an edge-switch serving as a gateway to a server cluster. The mechanism is primarily intended to be used as part of a content aware request distribution strategy. Our approach uses an extended form of network address translation which maps inbound connection information (ie. address, port, and sequence number) to a separate outbound connection. A key difference in our approach is that while the switch performs network address translation and TCP splicing, the actual hand-off is triggered by the back-end nodes. This relieves the switch of performing any application layer responsibilities. Nodes may hand-off connections by first initiating a new connection to the destination and then sending a message to the gateway which splices the two connections together. The gateway modifies subsequent packet headers in order to create a transparent hand-off. This mechanism requires no modification to the operating system on the servers or the clients and supports HTTP/1.1 persistent connections and pipelined requests. To test our design, we implemented a soft-switch using Linux Netfilter which includes the extended network address translation. We provide some preliminary performance analysis and make recommendations for future work.**

I. Introduction

The enormous growth of the Internet has created substantial workloads for Internet data centers. Content providers cope with the traffic by employing large clusters of rack-mounted servers. Google, a popular search engine and Internet portal, currently employs a cluster of 8000 machines and anticipates doubling the size of the cluster by the end of the year [5]. Google, like many other providers, prefers low-cost clusters of workstation-class systems to massive n-way SMP systems. Their choice is motivated by lower cost along with greater bandwidth to I/O devices. With the global Internet population slated to double over the next few years, the ability to rapidly scale services is a key factor for content providers.

While dynamic pages have become increasingly popular, the relative amount of static data as compared to dynamic data has remained constant [6]. There exists a whole market niche dedicated to serving this static content in the form of proxy caches, reverse proxy caches, and even geographically distributed static caches [1].

When serving static content, scalability is best achieved through distributing requests based on the content being requested. These mechanisms attempt to service requests on machines where the data is readily available, usually in memory or within the processor's data cache. This has lead to the development of layer-7 or application layer switching, where the edge-switch actually parses the incoming request and makes a decision about which back-end server to dispatch the request to. There have been several layer-7 web switches available in the market [3] [4] which provide a variety of services from load-balancing to cache redirection [7].

Resolving locality-based layer-7 routing problems for protocols such as HTTP at the switch is difficult. The incoming connection must first be accepted by the switch, the request received, parsed, and the route resolved to a back-end node. These operations do not scale well using typical switch hardware which is optimized for packet routing and forwarding. The switch must then perform some form of network address translation [17] or TCP Splicing [12] to map the initially accepted connection to the connection with the back-end. Furthermore, the connection must be constantly watched by the switch for new requests if intelligent request distribution is to be used for subsequent requests within a persistent HTTP-1.1 connection.

Therefore, we present a hybrid solution which we call *Knowledgeable Node Initiated TCP Splicing* (KNITS), in which the front-end switch maintains the responsibility of header manipulation and packet forwarding, but the actual layer-7 routing decision is made on the back-end nodes. This is accomplished by having the front-end switch spray the initial connection requests to back-end nodes in a load-balanced, round-robin fashion. A proxy application on the back-end node accepts the connection, parses the request, and resolves the dispatch decision. If a hand-off is called for, the back-end node opens up a connection to the target back-end node and then notifies the switch. The switch is responsible for splicing the initial connection to the newly opened connection and modifies subsequent packet headers appropriately to map the two connection states to one another.

As we will discuss later, a significant portion of the previous work in back-end distributed request distribution required sig-

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

nificant modification of back-end operating system kernels and applications. In contrast, we were primarily motivated by the desire to design a mechanism which could be used regardless of the architecture, operating system, or application running on the back-end server. To this end, we wanted to restrict system level modifications to the edge-switch servicing the cluster and limit the amount of application-level modifications necessary to use the mechanism.

Due to the fact that programmable switching hardware was not available to us, we implemented the prototype as a soft-switch. This prototype is primarily intended to be a proof of concept. The true performance and scalability potential of the mechanism will only be demonstrated through implementation on a programmable network processor.

The rest of this paper is structured as follows. Related work is described in Section II. Section III describes how the switch interacts with the TCP protocol stream to achieve hand-off. Section IV describes a prototype using a Linux box with a Netfilter extension as a soft-switch. Section V reports some preliminary performance benchmarks and analysis. We describe our proposal for the integration of our TCP hand-off mechanism into hardware switches and make some speculation about performance improvement based on hardware acceleration of packet network address translation and forwarding in Section VI, and we conclude in section VII.

II. RELATED WORK

Several approaches to content aware request distribution have been previously explored in the literature. A straightforward technique is to accept the connection at a front-end switch, which acts as a gateway for the server cluster, parse the request, and then forward the request to the back-end node best suited to handle it. The front-end handles the tasks of initial connection setup, request parsing, request dispatch, and setting up the connection to the back-end node. After the connection has been forwarded to the back-end node, the front-end still maintains responsibility for forwarding packets from the client to the back-end server and vice-versa. In a soft-switch implementation, the overhead of the forwarding can be somewhat mitigated by traditional TCP splicing [11]. An additional optimization is to use *pre-forked* connections to reduce the overhead imposed by the connection establishment between the front-end and the back-end server [27]. The overhead of establishing the initial connection, parsing the request, and resolving the back-end to whom the request should be dispatched is the bottleneck in software switch solutions [9] [27]. In a hardware switch the forwarding can typically be performed at wire speed even when doing network address translation. A hardware implementation for the switch [3] [4] [7] (layer-7 or application layer switching) leads to performance improvements at the cost of increased complexity and price. KNITS simplifies the complexity and workload of the front-end switch by removing from it the responsibility of the back-end server selection. Thus, a hardware implementation of KNITS will lead to improved performance with decreased cost and design complexity.

In order to reduce the workload of the front-end, an alternative approach was proposed in which connections are handed-off from the front-end to the back-end node allowing the back-ends to send responses directly to the client [9] [28]. This alleviates the network address translation of response packets and forwarding of response payloads, but ACK packets are still forwarded from the front-end to the back-end, and the front-end is still responsible for accepting initial connections, parsing requests, and making the dispatch decision.

To increase scalability, an improved design was suggested in which initial connections are sprayed to back-end nodes [10] spreading the overhead of connection establishment and request parsing among available nodes in the cluster, and ACKs are forwarded by a front-end soft-switch. Centralized dispatch was maintained in order to keep the solution simple and to limit control communication on the back-end network. Aversa and Bestavros suggested the use of round-robin DNS (RR-DNS) combined with a Distributed Packet Rewriting technique (DPR), in order to completely distribute the request distribution mechanism [24]. Their method makes unnecessary the use of a centralized connection router making it more fault tolerant. A major problem with traditional TCP hand-off solutions is that they require extensive modifications to the protocol stack of the nodes in the cluster. In practice these modifications are intimately tied to the implementation of the operating system protocol stack, do not track well with successive releases of the operating system, and are not easily portable to other UNIX variants. Another complication is that the above implementation [10] only allows TCP connections to be handed off once. After the initial hand-off, data for subsequent requests (as may be present in a HTTP/1.1 [2] persistent connection) is back-end forwarded.

Tang et al. proposed an implementation of TCP hand-off which makes unnecessary the use of a centralized front-end, while at the same time it remains portable, flexible, and transparent to the server application [26]. Their solution is based on the STREAMS based TCP/IP implementations, which are available on leading commercial operating system. STREAM-based TCP/IP offers the opportunity to implement the hand-off mechanism as a loadable kernel module which increases modularity and hence portability. KNITS and STREAMS-based TCP hand-off share similar goals. Both focus on providing an efficient mechanism for content-aware web request distribution while maximizing portability and transparency. The STREAMS-based implementation removes the centralized front-end at the expense of portability but it is based on an uncommon TCP/IP implementation and requires back-end kernel modifications.

An alternative is to spray requests through to back-end servers and back-end forward the requested data from a peer in

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

the event of a local-cache miss [8]. The data is passed through to the client without polluting the cache of the back-end with a replicated copy. This approach has a distinct advantage of simplicity and portability. With a sufficiently fast back-end network and an efficient back-end forwarding mechanism, performance is less than, but comparable to the performance achieved with TCP Hand-off. However, the use of specialized VIA gigabit Ethernet network hardware and software extensions limits the availability of this approach on standard hardware running standard application software.

Finally, Snoeren et al. suggested a set of techniques for providing fine-grained fail-over of long-running connections across a distributed collection of replica servers [25]. Their work focuses mostly on fault tolerant delivery of streaming media and telephony sessions. Their connection migration mechanism is based on the TCP Migrate Options [29], which require modifications in the protocol stack of both the client and server kernels.

Our proposed solution maintains portability by requiring only slight application level changes on the back-end. The use of a simple proxy application can provide hand-off facilities to existing legacy applications. Since connection establishment, request parsing, and request dispatch are still performed on back-end nodes, the switch itself does not become overly complicated and can be implemented very efficiently with hardware or hybrid hardware/software solutions.

III. Design

A. Connection Hand-off Using TCP Splicing

Network address translation, or NAT, maintains a mapping between one destination, represented by an IP address and port number, and another. This allows a connection destined for a certain IP address and port to be redirected to another address and port. This mapping is traditionally set up during the handshake when the connection is established. While NAT can be implemented in software, it has also been implemented within the firmware of intelligent layer-4 switches.

In order to accommodate connection hand-off, we must extend the mapping of address and port information to also map sequence and acknowledgment numbers. This is required as hand-off occurs after the connection has already been established, not during the initial handshake.

Special provisions need to be made to handle packets bound for a node prior to the hand-off. For example, say we have three nodes a client, server A, and server B. The client has an open connection with A, sending and receiving packets. Then A initiates a hand-off using TCP splicing to B. Now future packets from the client will be sent to B, but there may still be undelivered packets in the network such as acknowledgments and re-send requests which should be sent to A.

In order to accommodate this condition, the switch must maintain multiple translations for connections which are in the process of being handed off. Packets from the client with reference sequence numbers prior to the hand-off get routed to A, all other packets will be routed to B. This transient state will be garbage collected once the time to live window expires for undelivered packets (TCP-WAIT), and the connection to A from the switch can be closed.

Since handoff requests are initiated by back-end nodes, all network traffic among the back-end servers has to be intercepted by the front-end switch. For this reason, all packets initiated from the back-end nodes are routed through the switch, which makes the necessary modifications to the packet headers.

B. TCP Connection Hand-off Sequence

The splice request would typically be sent after the connection has been set up and the request has been received. It is at this point that an application proxy or the application itself can judge which node within the cluster is best suited to service the request. If the connection is to be handed off, the application layer sends the splice request to the switch. Multiple hand-offs can be accommodated on the same connection allowing a persistent session to migrate to multiple cluster nodes.

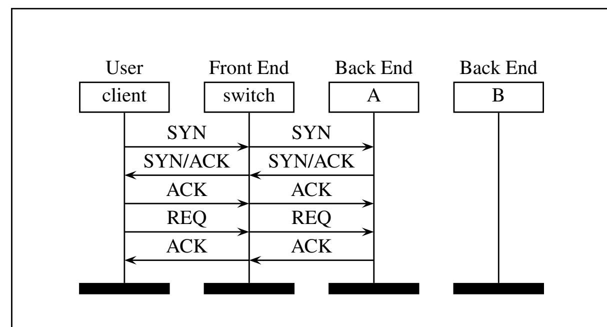

Fig. 1. Connection Establishment

1) Client opens connection (c1) w/SYN packet. Gateway round robin distributes connection request. Connection is accepted by Node A.
2) Client sends request to Node A.
3) Node A acknowledges the packet, parses the request and resolves request to be handled by node B.
4) Node A opens a connection (c2) to Node B with MSS equal to the client's MSS on the (c1) connection.
5) Node A sends hand-off request to switch and forwards original request to node B on (c2).
6) Switch receives hand-off request, splices (c1 $\longleftrightarrow$ c2) by updating network address translation matrix.
7) Node B receives forwarded request for Node A, acknowledges receipt of packet and begins to process it.
8) The switch sends FIN to Node A for (c1)
9) Node A shuts down (c1) and sends FIN/ACK.
10) Node A receives ACK from node B for request, and closes (c2) link. If the ACK for the request from node B

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

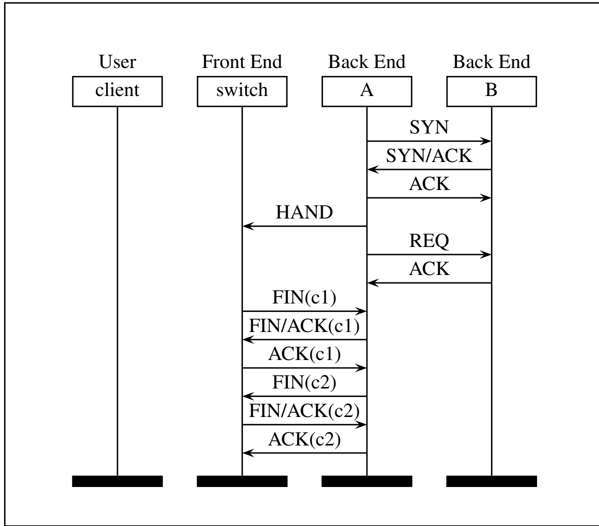

Fig. 2. Connection Hand-Off

had been piggy-backed with the response of the request, which was intercepted and forwarded to the client, the switch generates an ACK that is identical to the one that would have been generated by Node B and forwards it to Node A. The ACK is generated when the response packet is intercepted by the switch and an ACK packet for the request has not been seen (c2). The switch will intercept the FIN packet on (c2) and complete (c2) link shutdown with Node A without notifying node B.

11) Node B sends requested data to client using header state for (c2) connection. Since all server network traffic is routed through the switch, the packet will be intercepted by the switch and modified accordingly.

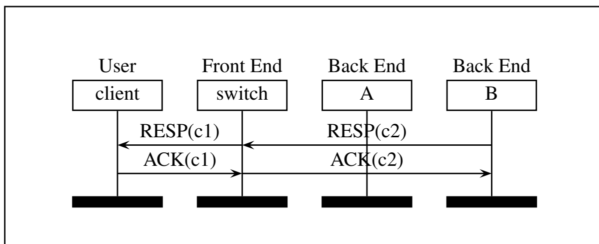

Fig. 3. Payload Header Manipulation

12) Switch modifies header information to map (c2) header to (c1) header and forwards packet to client.
13) Client receives packet and sends acknowledgment with (c1) header.
14) Gateway receives acknowledgment from client and maps (c1) header to (c2) header and forwards packet to Node B.
15) Node B receives ACK and continues data transmission. Data exchange continues in this fashion. If another request is received by Node B, it may initiate a second hand-off (go back to state 3).

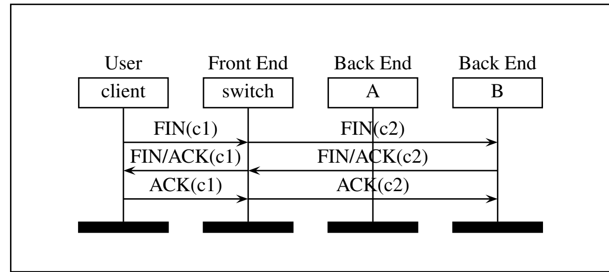

Fig. 4. Connection Cleanup

16) Client closes connection (c1). Gateway translates close request (FIN) and forwards to node B.
17) Node B receives close request and acknowledges it.
18) Client receives and acknowledges close request.
19) Gateway translates final close acknowledgment and forwards to Node B. After forwarding, the gateway removes translation from the matrix and cleans up state.
20) Node B receives close acknowledgment and cleans up state.

IV. Implementation

We chose to implement the prototype within the Linux Netfilter infrastructure. The Linux Netfilter [10] package encapsulates functional support for network address translation, firewalls, and other forms of programmable layer 4 and layer 3 filtering. The programming interface for the Netfilter package allows filtering based on IP address and port number of source and destination. In order to accommodate connection hand-off using TCP splicing, the Netfilter implementation must be extended in three ways.

First, the Netfilter architecture must be extended to allow transformation of sequence numbers. This is necessary in order to allow the gateway to map sequence numbers of connections in order to accommodate dynamic hand-off.

Second, the Netfilter architecture must also be able to filter based on sequence numbers. This will allow packets such as re-transmissions and acknowledgments to be delivered to Node B after a hand-off of the connection to Node C.

Third, the Netfilter architecture must be able to manipulate the TCP timestamp option included in most TCP packets. The TCP timestamp option allows the sender to calculate an RTT for each received ACK. It consists of two fields. The sender places its timestamp in the first field, while the second is filled with the timestamp of the last received packet. Since the clocks of different computing systems are not synchronized in general, Netfilter has to update the timestamp fields of every packet, after a hand-off in order to account for the clock differences. While this shouldn't be the case in well configured clusters, the timestamp mapping increases the overall robustness of the mechanism.

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

A. Linux Netfilter

Netfilter [18] consists of two main parts. First, each network protocol defines well defined points in a packet's traversal of that protocol stack, called "hooks". Second, kernel modules may register to listen to specific hooks of different protocols. When a packet traverses the protocol stack, Netfilter checks if any modules have registered for that protocol and hook, in which case they are given the chance to examine, possibly alter and provide a verdict for the packet. The verdict may be to *discard* the packet, *accept* which allows it to continue the traversal of the protocol stack, *steal* it, or request Netfilter to *queue* it for user-space processing.

IPv4 defines five hooks:

- *NF_IP_PRE_ROUTING* intercepts a packet when it is received by a node, before any IP processing or routing decisions take place.
- *NF_IP_LOCAL_IN* intercepts a packets that is destined for the local node, before the packet is processed by a higher level protocol and is passed to user-space.
- *NF_IP_FORWARD* intercepts a packet that is destined to a remote node.
- *NF_IP_LOCAL_OUT* intercepts a packet that is created locally, before any routing decisions take place.
- *NF_IP_POST_ROUTING* intercepts a packets immediately before it gets transmitted.

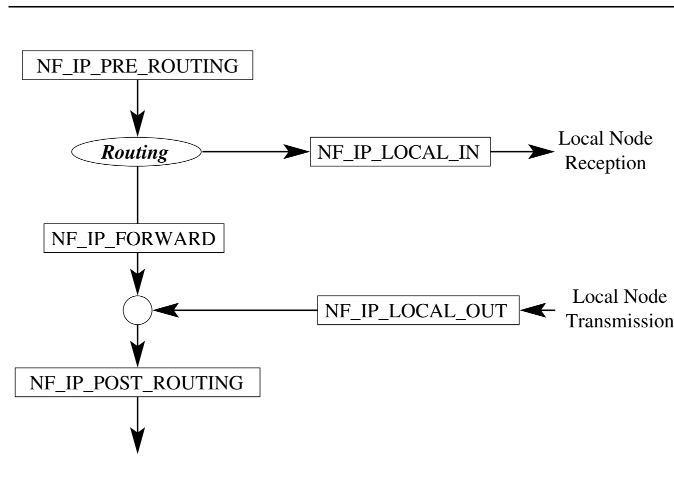

Fig. 5. Netfilter Hooks

The NAT module of Netfilter, on which the KNITS' implementation is based, uses the *NF_IP_PRE_ROUTING* and *NF_IP_LOCAL_OUT* in order to alter the destination addresses of passing-through and locally-generated packets respectively. Also, it uses the *NF_IP_POST_ROUTING* and *NF_IP_LOCAL_IN* for altering the source addresses of outgoing and locally destined packets. Through these mechanisms, the NAT module may be directed to intercept packets that match a specific protocol and Internet address pattern and alter their source and/or destination addresses.

Fundamental to NAT is the connection tracking module. The connection tracking module keeps track of all packets passing through the node, reassembles packets[1] and maintains necessary information about the status of each packet stream. Internally, it converts a packet into a tuple, representing the identifying parts of the packet. The tuple of TCP packets consist of the source address, source port, destination address, and destination port. The tuple for every packet in the same direction of the same packet stream stream is the same. For example all packets belonging to the same TCP connection from node *A* and port *X* to node *B* and port *Y* are represented by the tuple $< A, X, B, Y >$, while packets in the opposite stream of the same connection are represented by the tuple $< B, Y, A, X >$. The tuple of the first packet of a packet stream is its original tuple, while the tuple of a packet in the reverse direction is the stream's reply tuple. Tuples are used widely in Netfilter since they provide an indexing mechanism of the basic structures that maintain all information related to a specific stream.

Packet streams are represented by an *ip_conntrack* structure. Upon interception of the first packet belonging to a specific packet stream, the connection tracking module initializes an *ip_conntrack* structure for the stream and associates it with the stream's original and reply tuples. After the processing of the connection tracking module is completed, the packet is examined by the NAT module, which searches the NAT rules to determine if the packet's tuple and protocol matches those, for which NAT has to be performed. If the packet has to be altered, NAT updates the packet stream's identifying tuples to account for the modifications, makes the necessary changes to the packet's header. In addition to that, it saves the required modifications of both the original and the reply stream in the *ip_nat_info_manip* structure included inside the *ip_conntrack* structure representing the packet stream, so that the subsequent packets are altered automatically using the saved information, without having to search through the NAT rules for possible matches. Since NAT alters the destination and/or source addresses of packets, the tuples representing a packet stream have to modified accordingly to allow the correct association of future packets with packet stream information. For example, if TCP packets with the tuple $< A, X, B, Y >$ have to destined to node *C*, port *Z*, the reply tuple representing this connection will be changed to $< C, Z, A, Z >$ and all subsequent packets matching the tuple $< C, Z, A, Z >$ will be SNATed to $< A, X, B, Y >$.

B. KNITS Implementation

KNITS has been implemented as an extension of the NAT module of Netfilter. The NAT module intercepts at the prerouting hook all packets that pass through the font-end node

[1] Reassembling packets is necessary for correct Network Address Translation since part of a packet's Internet address is included in the transport protocol header which can be found only in the first fragment of a fragmented packet.

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

and examines their destination address[2]. If the address matches the well-known front-end address, NAT alters the packet's destination address to the address of a back-end server and updates the *ip_conntrack* structure. Two major changes have been made in the NAT implementation.

First, since KNITS requires the tracking of sequence numbers and timestamps, NAT has been modified to record the sequence number, the acknowledgment number and the timestamps of all intercepted packets. The *ip_nat_info_manip* structure has been extended to maintain various possible NAT bindings for each packet depending on the packet's sequence number. To achieve this, two fields representing a minimum and a maximum sequence number have been added to the structure. The manipulations recorded within the structure are applied to a packet only if the packet's sequence numbers fall within the range of sequence numbers indicated by the minimum and maximum value.

Second, a kernel thread listens constantly to a well-known UDP address for hand-off requests. A hand-off request may be generated by a back-end server and includes: the client's address and port, the server's address and port and the target server's address and port. This information allow the KNITS thread to construct the tuples that represent the two connections (client-server and server-target) that are associated with the hand-off. Upon reception of a hand-off request the kernel thread updates the *ip_nat_info_manip* structure corresponding to the connection for which the hand-off was requested, so that new client packets will be DNATed to the target server and SNATed to the initial server from which the target server expects requests, and replies of the target server will be DNATed to the client and SNATed to the well-known front-end address, from which the client expects to receive replies. In addition to that it computes, the difference between the two server's sequence numbers, acknowledgment sequence numbers, and the servers' clocks. These offsets are used to correct the TCP header's sequence number and timestamp fields of the target's TCP packets to those values that the client expects to see. Similarly, the difference between the client's and the target server's sequence numbers and clock is calculated in order to correct the subsequent client's packets, that are DNATed to the target server, to those values that the target server expects to see. These offsets are saved in the *ip_nat_info_manip* structure with the rest of the manipulations. The reason for this is that packets received by the client and the target server have to look as if they had been sent by the initial server.

Until the initiation of a hand-off and after its completion, the modified NAT module operates in the same way as the original one, with the exception that it records packet sequence number and timestamps and that it checks a packet's sequence number in order to find the appropriate manipulations for the packet.

[2]Fragments of fragmented packets are collected by the connection tracking module. Upon reception of all fragments the packet is re-assembled and gets injected into the protocol stack.

However, while a hand-off is in progress, certain packets require special handling. In order for the hand-off to complete the following packets have to be seen through the connections associated with the hand-off, and processed by KNITS:

1) The client's request has to be sent from the initial server to the target server. Upon receiving the request, the switch creates a FIN packet similar to one that would have been generated by the client, and transmits it to the initial server, in order to terminate the initial server's connection to the client. The acknowledgment and the FIN/ACK packets, with which the server will respond will be intercepted by the switch and be dropped. Upon the reception of the FIN/ACK packet, the switch will generate an ACK packet, send it to the server, and hence complete the 4-way connection termination protocol.
2) An ACK of the request will be sent from the target to the initial server. This ACK packet is forwarded to the initial server without any modifications.
3) The response to the request will be transmitted from the target server to the initial server along with a piggy-backed ACK for the forwarded request. The switch intercepts the packet and forwards it to the client along with generating an ACK which is sent to the initial server if one has not already been sent.
4) After forwarding the request the application-level proxy running at the initial server closes the connection to the target server. Thus, a FIN packet will be sent from the initial server to the target server, after the transmission of the request. The switch has to drop this packet, so that it never reaches the target server. In addition, it has to generate and transmit a FIN/ACK packet that imitates a FIN/ACK sent from the target to the server.
5) An ACK packet, that acknowledges the reception of the target's FIN/ACK will be sent from the initial server to the target. The switch will intercept and drop the packet.

Hand-offs subsequent to the first are processed in a similar way. A small complication that arises is that subsequent hand-off requests do not include the address of the client. For a secondary server, its client is the address of the server that handed the connection to it. Thus, the tuple that represents the original connection from the client to the server cannot be reconstructed. To solve this problem, a pointer to the original client-server connection has been added inside the *ip_conntrack* structure describing every connection hand-off, allowing the discovery of all information necessary to complete the hand-off.

C. Application Layer Code

Our primary motivation was to minimize changes required on the back-end servers. To this end we designed and implemented KNITS in such a way that required no kernel or application modifications on these servers. However, an application must initiate hand-off in order for the mechanism to be used.

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

We chose to implement this functionality in a proxy application, which accepts the initial connection, parses the request, resolves who should handle the request, and then either dispatches the request to the local web server or hands the connection off to a peer. The details of dispatch resolution are beyond the scope of this paper. Several approaches to dispatch resolution have been proposed in the literature [9] [10] [24]. Requests are handed off to the application-layer web-server using the same mechanisms as TUX [30], the Red-Hat kernel-level static web-server.

Two special cases must be accommodated by the proxy application in order to handle HTTP/1.1 [2] correctly. Persistent connections, as defined in section 8.1 of the aforementioned specification, allow clients to use a single connection for multiple requests. Furthermore, clients may "pipeline" their requests, sending all requests at once without waiting for a response. The proxy application accommodates these situations by holding on to the connection and processing each request individually and in-order. When a request in the "pipeline" is not best handled on the local node, the proxy application initiates a hand-off and forwards all remaining requests to the peer node. The ability for KNITS to perform multiple hand-offs per connection is something which distinguishes it from previous solutions which only support an initial hand-off and then use back-end forwarding [10] for subsequent requests on the same connection.

V. Performance

We focused our performance analysis on a set of micro-benchmarks whose primary purpose was to get an idea of the performance penalty paid by the different portions of the KNITS mechanism for a single client and a single server thread. The test was a simple serialized request/response, where a new request would not be sent until the response had been received in its entirety. We took measurements of throughput, latency, and connections per second using this model. For the throughput and latency tests, multiple requests were sent over the same connection. For the connections per second test, a single request was sent per connection. All workloads were served directly from memory, so disk activity was not a factor in the experiments.

The experiments used a dual-processor 866 MHz Pentium III system with two Alteon Gigabit Ethernet NICs and 1 gigabyte of 133 MHz SDRAM for the soft-switch. The switch was running a standard Linux 2.4.4 kernel with an extended form of Netfilter enabled. The MTU size of the switch was set to 1500. The micro-benchmarks used a similar system for the client and the back-end servers. All systems were interconnected using an Extreme Black Diamond gigabit Ethernet switch.

Six different tests were performed in an attempt to assess the performance impact of the various components:

- *Direct*: This was the base-line for the experiment. The test was run on two machines interconnected directly with a cross-over cable. No hardware or software switch is present. The results of this test should represent the best possible numbers for the configuration and test software we were using.
- *Forward*: In order to test the cost of forwarding packets through a standard Linux kernel, we placed a system which we will call the soft-switch in between the client and the server. All packet traffic between the client and the server were routed through this soft-switch. The standard IP forwarding mechanisms within the Linux kernel are used to route packets between the two interfaces.
- *Netfilter*: We then enabled Netfilter with SNAT and DNAT translation on the soft-switch so that we could measure the cost of header manipulation and checksum computation incurred by the NAT mapping of packets.
- *KNITS*: To measure the impact of tracking the additional information during connection setup that KNITS requires, we performed a test which ran through the KNITS infrastructure with none of the requests triggering a hand-off.
- *KNITS-Handoff-1*: The performance impact of hand-offs was measured by performing tests in which the first request of each connection triggered a hand-off.
- *KNITS-Handoff-2*: To see if subsequent hand-offs incurred any additional penalty versus a single hand-off, we ran a test, in which two hand-offs were triggered by the requests of each connection.

The progression of tests decompose the various components of the KNITS mechanism: test application overhead, packet forwarding, network address translation, connection tracking, hand-off, and multiple hand-off.

Figures 6 and 7 present the client perceived throughput and latency for each of the six scenarios. All micro-benchmark experiments were conducted seven times. The lower and higher results of each experiment are ignored and the average of the remaining five values is presented in this section.

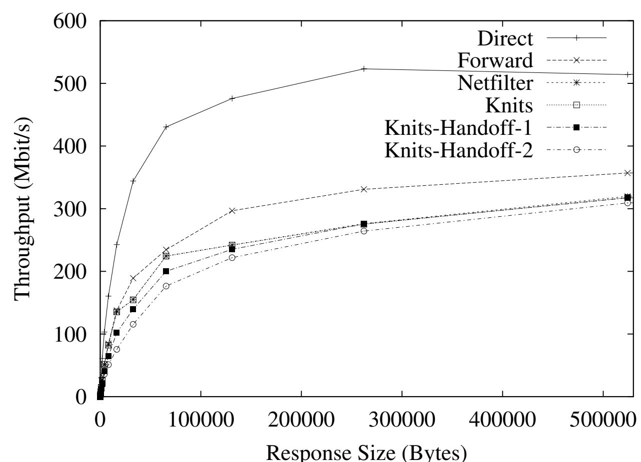

Fig. 6. Throughput Micro-benchmark

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

The first micro-benchmark was intended to measure throughput limitations incurred by the use of the hand-off mechanism. The throughput measured is based on the amount of data in the response packets, not bi-directional throughput. A direct connection achieves a maximum throughput of 580 Mbit/s with a response size up to 32 MB. Using a soft-switch, through which packets are forwarded, reduces the maximum achievable throughput by 33% to 385 Mbit/s. Doing network address translation to forward packets through the soft-switch either with Netfilter or KNITS reduces the maximum throughput to 377Mbit/s, a 35% reduction when compared to the direct connection. In the presence of hand-offs, the maximum throughput is reduced by 36% to 371 Mbit/s for both scenarios (one or two hand-offs per connection).

The throughput measurements essentially show that KNITS scales reasonably to around 64k response sizes. After this point throughput levels out quickly. Also, it maintains a comparable throughput to any form of network address translation – even for multiple hand-offs.

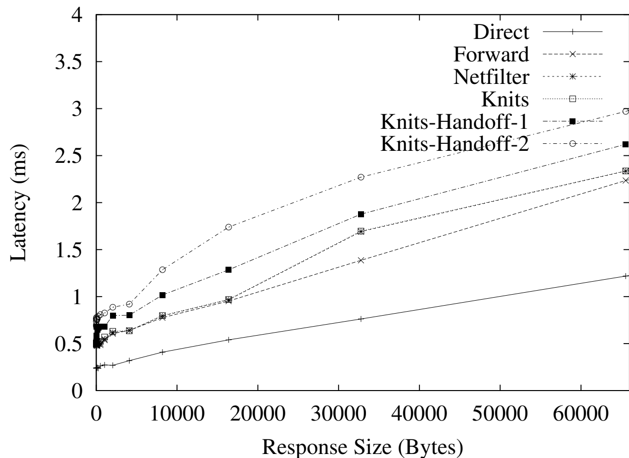

Fig. 7. Latency Micro-benchmark

The latency micro-benchmark is measured from the client's perspective from the time the request was sent until the time the entire response is received.

The latency results again show that most of the performance penalty is due to the overhead of forwarding the packets. For response sizes of 64 KB the client perceived latency was 1.22ms for the direct connection, 2.24ms for the forwarding scheme, 2.33ms for Netfilter and KNITS without hand-offs, 2.62ms for KNITS with 1 hand-off and 2.97ms for KNITS with 2 hand-offs.

The latency graph also shows an average latency increase of approximately 500 microseconds per hand-off. This is most likely due to the cost of establishing a new connection between the two back-end nodes. The service time of subsequent requests remains unaffected.

When comparing KNITS or Netfilter with the Direct connection results, nearly all the additional overhead is comprised of the time to forward the packet through the soft-switch. The next largest performance impact is due to TCP/IP header manipulations. Both Netfilter and KNITS without hand-offs require only modifications to the address fields of the IP and TCP header, and a partial re-computation of the TCP and IP checksums, and hence they exhibit equivalent performance. In the presence of hand-offs, additional modifications are required for the sequence number fields and the timestamp option of the TCP header. Most of the performance difference between KNITS without hand-offs and KNITS with hand-offs occurs because of the additional messages that have to be exchanged in order to complete the hand-off. Requests that are subsequent to a hand-off are serviced with similar latency as requests prior to the hand-off.

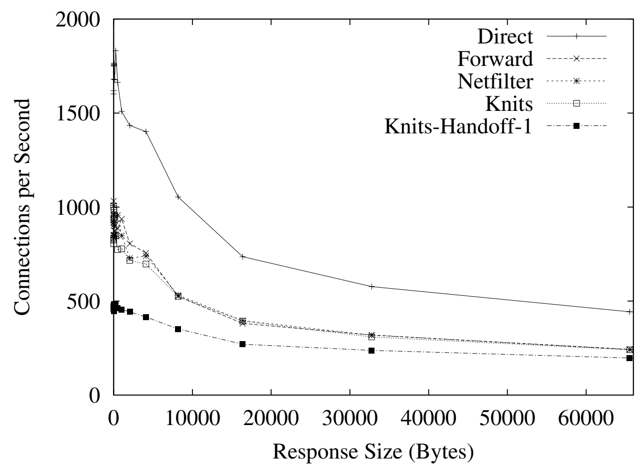

Fig. 8. Connections Per Second Micro-benchmark

The connections per second benchmark sends a single request per connection. A connection is opened, the request is sent, the entire response is received, and then the connection is closed. A new connection is not opened until the entire response is received and the connection is closed.

While the connections per second results show that forwarding is still the primary bottleneck, there is also a dramatic drop in performance for connections which have a hand-off for small packet sizes. This is due to the serialized nature of the micro-benchmark which doesn't reflect a scalability bottleneck in the mechanism. For a response size of 64KB, the direct connection achieves 443 connections per second, the forwarding scheme exhibits 243 connections per second, Netfilter and KNITS 241 connections per second, and KNITS with a single hand-off 197 connections per second.

VI. Future work

The Netfilter implementation, while providing the facilities to rapidly prototype the design, exhibits poor performance

primarily due to the copies introduced by packet forwarding through a standard Linux kernel. Additionally, the Netfilter infrastructure adds a lot of unnecessary overhead to the forwarding process. A soft-switch prototype would be better implemented in an infrastructure designed for high throughput such as MIT's Click Modular Router Project [21]. Among other things this would remove copies from the forwarding process and improve performance through intelligent event notification and streamlined packet handling. Recent versions of Click have shown good scalability on SMP systems with performance approaching 500,000 packets/second [22]. Further improvements could be made by utilizing intelligent gigabit NICs with payload cache firmware [23].

Despite the potential improvements to the soft-switch architecture, it is our belief that the only truly scalable solution will be achieved using hardware acceleration features as found in modern network processors. Therefore, we view the next step of development to be implementation of the KNITS NAT extensions using the parallel pipeline pico-engines present on the IBM NP4GS3 network processor [16].

The IBM PowerNP reference design provides the hardware and software components for a high-capacity programmable network switch. The network processors have specialized hardware well suited to performing translation table look-ups as well as the sort of header manipulation required for extended network address translation. The reference hardware is capable of 28.4 GB/s throughput, 4.5 million packets per second (which implies 750,000 conns/s according to [10].

Since forwarding occurs at line rate within the network-processor, the only overhead involved for our implementation of TCP hand-off would be associated with the table look-up to retrieve mapping information, and the cost of the actual header manipulation. Since, most of the overhead of header manipulation is the re-computation of checksum values and the PowerNP has a co-processor to which this task can be off-loaded to, we anticipate a minimal performance penalty for the extended network address translation. As shown in our benchmarks of the soft-switch, this relatively insignificant overhead diminishes as packet sizes increase.

We have primarily talked about connection hand-off in terms of locality aware request distribution. However, it is clear that with some modifications the mechanism could be used for connection take-over as well as connection consolidation for long-lived requests such as media streaming. We would like to investigate what changes would be required within the mechanism to allow connection take-over by a peer, and investigate what impact this would have on the performance and robustness of the system.

VII. Conclusion

We have laid out the motivation and design of a mechanism for performing TCP hand-off at an edge switch acting as a front-end to a server cluster. We have described a prototype soft-switch implementation using an extension of the standard Linux Netfilter package, and done some preliminary performance testing. The Netfilter extensions required by KNITS were implemented in a single file of 2200 lined of code including comments.

Our results prove that this mechanism indeed functions as intended, and its robustness was verified through thousands of requests performed during the duration of our tests. The implementation of the application level proxy was a mere *600* lines of code, which were the only modifications necessary on the back-end servers.

Three important conclusions from the micro-benchmark experiments are:

1) Most of KNITS' performance loss when compared to a direct connection is because of the overhead of forwarding over a soft-switch. KNITS accounts for only a 2-3% additional throughput loss when compared to a normal forwarding scheme on a soft-switch.
2) KNITS performs similarly to the standard Netfilter NAT mechanism in the absence of hand-offs.
3) In the presence of hand-offs, only the service time of the request that causes the hand-off is affected. Requests prior and subsequent to the hand-off are serviced with the same latency.

These observations support our general claim that if the KNITS NAT extension was implemented within the firmware of a programmable layer-4 network switch we would achieve a scalable, portable solution to intelligent request distribution and connection hand-off.

References

[1] Akamai Technologies, Inc., *Akamai freeflow service,* http://www.akamai.com/service/network.html

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee *Hypertext Transfer Protocol – HTTP/1.1,* Internet RFC 2068, January 1997.

[3] Cisco Systems, *Cisco Content Services Switch,* http://www.cisco.com/univercd/cc/td/doc/product/webscale/css/

[4] Alteon Websystems, *ACEDirector,* http://www.alteonwebsystems.com. Oct 1998.

[5] M. Wagner, *Google Defies Dotcom Downturn,* Internet Week, April 2001.

[6] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, H. Levy, *On the scale and performance of cooperative Web proxy caching,* In Proceedings of the 17th ACM Symposium on Operating Systems, Principles(SOSP'99), March 1999.

[7] *Web Switches Shoot-Out,* PC Magazine Labs Report, April 2000.

[8] E.V. Carrera, R. Bianchini, *Efficiency vs. Portability in Cluster-Based Network Servers,* Rutgers University, In Proceedings of ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, June 2001.

[9] V.S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, E. Nauhum, *Locality-Aware Request Distribution in Cluster-based Network Servers,* Rice University, In Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, Oct 1998.

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

[11] O. Spatscheck, J.S. Hansen, J.H. Hartman, L.L. Peterson, *Optimizing TCP Forwarder Performance,* Technical Report TR98-01, University of Arizona, 1998. Published in the IEEE/ACM Transactions on Networking, 2000.

[12] A. Cohen, S. Ramgarakam, H. Slye, *On the Performance of TCP Spicing for URL-aware Redirection,* In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, CO, Oct 1999.

[13] W. Zhang, *Linux Virtual Server for Scalable Network Services,* National Laboratory for Parallel and Distributed Processing, Changsha, Hunan, China, Publish in Ottawa Linux Symposium, 2000.

[14] IBM. *Netdispatcher: A TCP connection router,* http://www.ics.raleigh.ibm.com/netdispatch/, May 1997.

[15] D.A. Maltz, P. Bhagwat, *TCP Splicing for Application Layer Proxy Performance,* IBM Research Report RC 21147, 1998.

[16] IBM Microelectronics Division. *The Network Processor: Enabling Technology for High-Performance Networks,* IBM Microelectronics Division, Hopewell Junction, NY, 1999.

[17] K. Egevang, P. Francis, *The IP Network Address Translator (NAT),* Internet RFC 1631, May 1994.

[18] H. Welte, *The Netfilter framework in Linux 2.4,* Sept 2000.

[19] T. Spalink, S. Karlin, L. Peterson, *Evaluating Network Processors in IP Forwarding,* Technical Report TR-626-00, Princeton, NJ, Jan 2001.

[20] Intel Corporation, *XP1200 Network Processor Datasheet,* Sept 2000.

[21] R. Morris, et al., *The Click Modular Router,* In 17th Symposium on Operating Systems Principles, 1999.

[22] B. Chen, R. Morris, *Flexible Control of Parallelism in a MultiprocessorPC Router,* Published in the proceedings of the USENIX annual technical conference, June 2001.

[23] K. Yocum, J. Chase, *Payload Caching: High-Speed Data Forwarding for Network Intermediaries,* Duke University, Published in the proceedings of the USENIX annual technical conference, June 2001.

[24] L. Aversa, A. Bestavros, *Load Balancing a Cluster of Web Servers Using Distributed Packet Rewriting*, Published in the proceedings of the IEEE International Performance, Computing, and Communication Conference (IPCCC'2000), Phoenix, AZ, February 2000.

[25] A. C. Snoeren, D. G. Andersen, H. Balakrishnan, *Fine-Grained Failover Using Connection Migration*, Published in the proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01), pp. 221-232, San Francisco, CA, March 2001.

[26] W. Tang, L. Cherkasova, L. Russell, M.W. Mutka, *Modular TCP Handoff Design in STREAMS-Based TCP/IP Implementation*, Published in the proceedings of the First International Conference on Networking (ICN-2001), Colmar, France, July 2001.

[27] C.S. Yang, M.Y. Luo, *Efficient Support for Content-based Routing in Web Server Clusters*, Published in the proceedings of the 2nd USENIX Symposium on Internet Technologies (USITS'99), Boulder, CO, October 1999.

[28] M.Y. Luo, C.S. Yang, *Constructing Zero-loss Web Services*, Published in the proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (Infocom'2001), Anchorage, AK, April 2001.

[29] A.C. Snoeren and H. Balakrishnan, *An end-to-end approach to host mobility*, *Constructing Zero-loss Web Services*, Published in the proceedings of the 6th Annual International Conference on Movile Computing and Networking (Mobicom'00), Bostom, Ma, August 2000.

[30] Red Hat, Inc. *TUX 2.1: Reference Manual* April 2001.

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.