Design, Implementation and Performance of a Content-Based Switch

G. Apostolopoulos D. Aubespin V. Peris P. Pradhan D. Saha
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
{georgeap,aubespin,vperis,prashant,debanjan}@watson.ibm.com

Abstract— **In this paper, we share our experience in designing and building a content based switch which we call L5. In addition to the layer 2-3-4 information available in the packet, a content based switch uses application level information to route traffic in the network. Making routing decisions based on information contained in the payload is not a new idea. In fact application level proxies which are functionally equivalent to a content-based switch, have been around for years.**

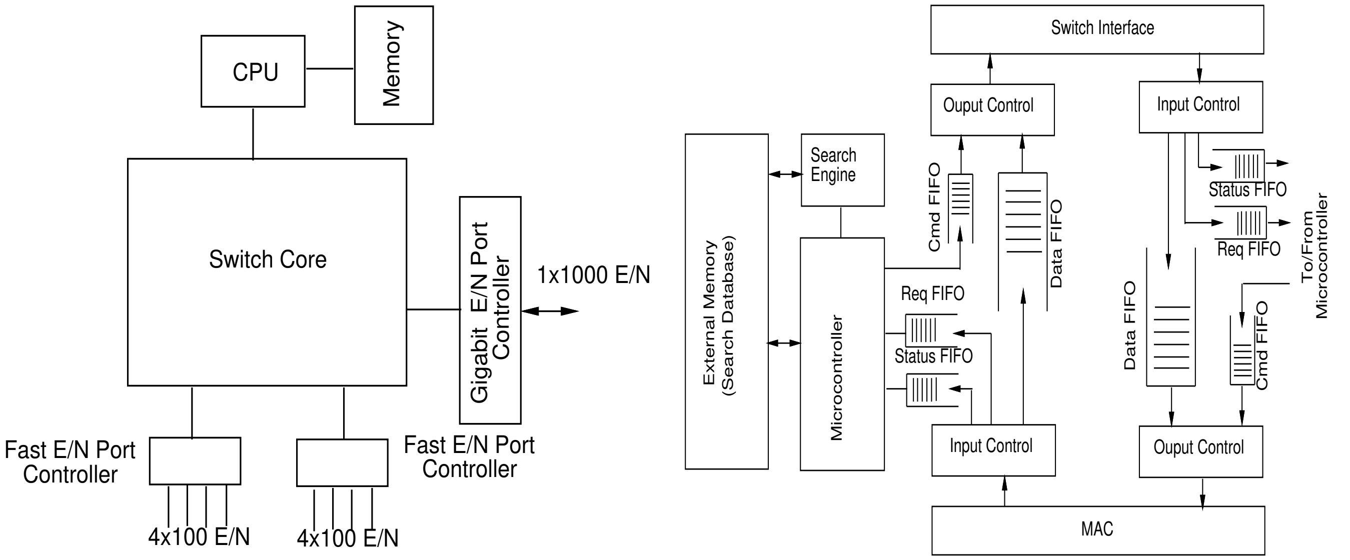
Our contribution is in combining the functionalities of an application level proxy with the data handling capabilities of a switch into a single system. In this paper, we describe the architecture of the L5 system along with the details of how application level information can be efficiently processed in switch hardware. We cover two specific application examples that we believe are ideal candidates for content-based switching: one is routing HTTP sessions based on Uniform Resource Locators (URL) and the other is session-aware dispatching of Secure Socket Layer (SSL) connections.

I. Introduction

Until recently the word switching was synonymous with forwarding frames based on link layer addresses. Of late, the definition of switching has been extended to include routing packets based on layer 3 and layer 4 information. Layer 3 switches, also known as IP switches, use IP addresses for network path selection and forwarding. They are fairly commonplace today and are being used as replacements for traditional routers. In addition to layer 2 and 3 information, a layer 4 switch examines the contents of the transport layer header, such as TCP and UDP port numbers, to determine how to route connections. Layer 4 switches are slowly making their way into the marketplace and are primarily used as load balancing connection routers for server clusters. Moving one level higher in the protocol stack, we can define a layer 5 switch that uses session level information, such as Uniform Resource Locators (URL) [5], in addition to layer 2-3-4 information to route traffic in the network. In this paper, we share our experience in designing and building a layer 5 switch, which we call L5. Although, the L5 system can potentially be used anywhere in the network, we mainly focus on its usefulness as a front-end to a server cluster. We explore the value of a content aware session router in a cluster of Web servers and Web caches.

Web server and Web cache clusters [3] are commonplace in many large Web provider sites and Internet Service Provider (ISP) GigaPOPs. In a typical installation, the server and the cache clusters are front-ended by layer 4 switches which distribute connections across the nodes in the cluster [7]. The main objective of these layer 4 switches is to balance load among the servers in the cluster. Since, layer 4 switches are content blind, this approach mandates that the nodes in the Web server cluster are either completely replicated or share a common file system. Besides the storage overhead, complete replication is also an administrative nightmare since every time a page is updated it has to be propagated to all the nodes in a relatively short period of time. A shared file system, on the other hand, increases the load on the server nodes as they have to first obtain the file from the file server before serving it out to the client. In a Web cache cluster, load balancing without considering the requested URL, leads to cluster nodes becoming redundant mirrors of each other. The L5 system takes into account session level information, such as URLs when routing a connection to a server node. Consequently, it makes it possible to partition the URL space among the server nodes thus improving the performance of the server cluster. As a session aware load balancer for a Web cache cluster, the L5 system effectively partitions the URL space among the cluster nodes, thereby increasing the total amount of content that is cached and improving the performance of the cache cluster.

A layer 5 switch can be a valuable tool when it comes to distributing secure sessions among a cluster of secure Web servers. Secure HTTP [4] sessions use the Secure Socket Layer (SSL) [8] protocol for privacy and authentication. The SSL protocol involves a computationally expensive handshake procedure that enables the client and server to authenticate each other and share a secret. Once this is done, subsequent SSL sessions can be easily setup using the same shared secret to generate symmetric keys for each session. A content blind dispatching of SSL sessions results in different sessions being dispatched to dif-



(a) Switch Architecture.

(b) Port Controller Architecture.

Fig. 1. Switch & Port Controller Architectures.

ferent nodes in the cluster. Since the server nodes do not share their secrets, content blind dispatching forces the client and the server to perform handshake operations for almost all sessions. This is computationally expensive and significantly reduces the number of connection requests the server cluster can handle. We show that the L5 system can greatly improve the overall throughput of a secure Web server cluster by dispatching SSL connections based on session information.

While layer 5 switching is exciting and useful, to date very few high-performance layer 5 switching systems have been built. One of the reasons why layer 5 switches are so rare is that most layer 5 protocols are designed to be handled by general purpose CPUs at the end-hosts, and typically involve complex protocol processing. A more subtle, but probably more compelling reason why switching based on layer 5 information is difficult, is an artifact of the TCP [9] state machine. Most layer 5 protocols that are of interest to us run on top of TCP. In order for a layer 5 switch to obtain the session level information necessary to perform content based switching, it first has to establish a TCP connection to the source. Once the connection has been established, migrating it from the switch to the destination is an extremely difficult task. In [13] the authors propose a technique to migrate live TCP connections. Unfortunately, their solution requires modifications to the TCP state machine and the message format. Given the large installed base of clients and servers using TCP and the reluctance of the vendors to modify Operating System kernels, any solution requiring mandatory modifications to the TCP stack is of limited practical value.

Application level proxies [10], [11], which are in many ways functionally equivalent to layer 5 switches, use a different approach to avoid migrating TCP end-points. They establish two TCP connections – one to the source and a separate connection to the destination. The proxy works as a bridge between the source and the destination, copying data between the two connections. While application layer proxies are functionally rich and flexible, they can not handle the high volumes of data that a switch is expected to handle. Our objective in designing the L5 system has been to combine the functionalities of application layer proxies and the data handling capabilities of switches into one system. Another salient feature of the L5 system is that it requires minimal configuration. Using two application examples we describe how the L5 system automatically learns layer 5 routing information, thus minimizing the need for configuration.

The rest of the paper is organized as follows. In section II we describe the hardware and software architecture of the L5 system. In section III, we describe the layer 4 processing required to support layer 5 functions. Section IV is devoted to URL based HTTP session routing and its performance. In section V, we describe how an L5 system can be used to balance the load of secure HTTP sessions that use the SSL (Secure Socket Layer) protocol. We conclude in section VI.

II. Switch Architecture

A high level illustration of the L5 system is shown in Figure 1(a). As shown in the figure, the L5 system consists of a switch core to which a number of custom built intelligent port controllers are attached. In addition, the L5 system is equipped with a processor complex. Layer 5 functions, such as the parsing of HTTP protocol messages and URL based routing, are performed by the processor. The job of the port controllers is to identify the packets

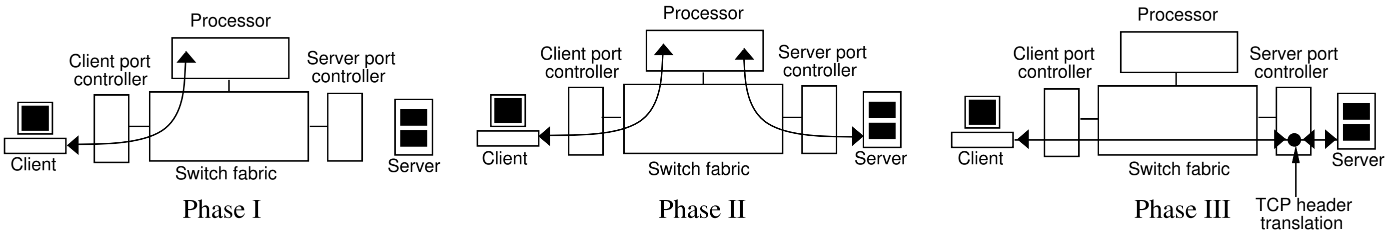


Fig. 2. Example flow through a Layer 5 Switch

that require layer 5 processing and forward them to the processor. In our design, we make sure that only packets that need to be handled by the processor are forwarded to it. The rest of the packets are processed by the port controllers. As we will see later in the paper, in most common scenarios only a very small fraction of the packets are processed by the CPU. As a result we can achieve very high speeds while delivering sophisticated layer 5 functionality.

The switch fabric consists of a shared memory cell/packet switching elements that is scalable from 2.5Gbps to upto 40Gbps. Figure 1(b) shows a simplified architecture of a port controller. The multi-tasking microcontroller is capable of processing multiple packets simultaneously. The search engine implements longest prefix match in hardware.

The microcontroller polls the request FIFO and picks up the packets waiting in the data FIFO for processing. As a part of packet processing, the microcontroller can extract any part of the packet from the data FIFO. In our system, we use the microcontroller for layer 2-3-4 processing. This involves extracting the layer 2,3, and 4 headers from the data FIFO, composing search keys using different parts of the headers, performing one or more searches using the search engine, and then executing necessary actions based on the search results. To be able to forward packets at wire-speed we have a processing budget of about 300 instructions, assuming a packet size of 64 bytes and a link speed of 1 Gbps. This is sufficient for our purpose as there is also a search engine that runs as a co-processor of the microcontroller.

The processor complex is a PowerPC 603e which is attached to the CPU port of the switch. The software running on the CPU is responsible for control functions as well as layer 5 data-path functions which are described in Section III. At the time of switch initialization, the processor downloads the microcode to the port controllers. The search databases are also initialized at this time. All layer 2,3, and 4 data path processing is handled by the port controllers. Packets requiring layer 5 processing are identified by the port controllers and are forwarded to the CPU for further processing. In the next section, we discuss how the CPU and the port controllers coordinate to implement layer 5 switching.

III. Operational Blueprint

The basic working principle of the L5 system is similar to that of an application layer proxy. It involves three major steps as shown in Figure 2. First (phase I in Figure 2), it intercepts the TCP connection setup request from the client and responds by establishing a connection to the client. It acts as a proxy for the server reading in as much layer 5 information as is needed to make a routing decision. Depending on the specific layer 5 protocol involved, it parses the layer 5 protocol messages and determines where to route the session based on the corresponding layer 5 routing database. After the routing decision is made, it sets up a second connection to the appropriate server node (phase II in Figure 2). In an application layer proxy, the processor remains on the data path and copies data between the two connections. In the L5 system, the processor gets out of the data path at an opportune moment by splicing the two TCP connections. After splicing (phase III in Figure 2), all packet processing is handled by the port controllers leading to very efficient data handling through the switch. Notice that the splicing of the connections requires TCP sequence number translation at the port controllers. Hence, although the port controllers do not perform any layer 5 functions, they play a very critical role in ensuring that layer 5 switching is fast and efficient. In the rest of this section, we discuss in detail how the CPU and the port controllers work in harmony to execute the three phases of layer 5 switching shown in Figure 2. Specific examples of layer 5 protocol processing are discussed in Sections IV and V.

A. *Processing at Port Controllers*

To better understand the working principles of the L5 system, let us consider the example scenario shown in Figure 3. In this setup, the L5 system is used as a front-end to a Web server cluster. The responsibility of the L5 system is to route HTTP requests from the clients to the nodes in the cluster based on the URLs in the requests. As is typical of many configurations of this nature, all nodes in the cluster share a common IP address, say VIP, and are known to the external world through this address. Additionally, each node in the cluster also has its own unique IP and MAC addresses. Let us assume that the L5 system

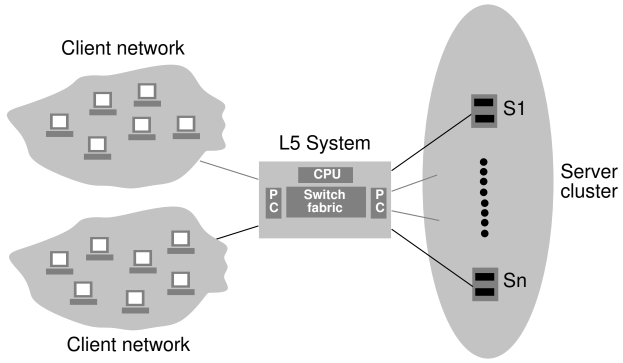


Fig. 3. Typical scenario in which the L5 system is a front-end to a server cluster

is configured to perform layer 5 switching on all TCP connections with destination address VIP and destination port 80 (default HTTP port).

Packet processing at the port controllers to which servers are connected is different from that at other ports. As a result, we distinguish server ports from other ports which we refer to as client ports. When a packet arrives at a port controller it is passed through a classifier. The classifier is responsible for identifying the processing needs of the packet based on layer 2,3, and 4 header information. Associated with each classifier is one or more action flags and necessary meta information that determines how the packet is processed. Tables I and II show the classification tables at the client and server ports, respectively, for the example discussed below. Notice that there are two types of classifiers – permanent and temporary. Permanent classifiers are installed during the switch initialization time and are not deleted unless there is a change in configuration. Temporary classifiers are installed and removed as connections come and go. Each classifier has a priority level associated with it. In the event of a conflict, the classifier with the highest priority overrides the lower priority classifiers. In our example P_i denotes a higher priority than P_j, if $i > j$.

Let us now consider a HTTP session originating from client address CA and client port CP and destined to address VIP and port 80. Figure 4 shows the timing diagram of the different steps involved during the connection setup as well as the data transfer phases. The client initiates the connection by sending a TCP SYN packet. In addition to setting the SYN control bit in the TCP header, the client also chooses a 32-bit starting sequence number (CSEQ) which is used to keep track of the data that the client sends to the server. This SYN packet makes its way to the client port where it is trapped by classifier CC1 (in Table I) and is forwarded to the switch CPU (step 1 in Figure 4).

The CPU receives the packet (step 2 in Figure 4) and then responds (step 3 in Figure 4) with a SYNACK message masquerading as the server. It uses VIP:80 as the source address and port number in the SYNACK packet. The starting sequence number (DSEQ) is chosen as *DSEQ = H(CA,CP)* where H is a suitable hash function that returns a 32-bit number, and CA, CP are the client IP address and port number, respectively. The hash function H is known to all the port controllers allowing them to independently compute $DSEQ$ given CA and CP. This eliminates the need for the CPU to send a separate control message with sequence number information to the port controllers.

The SYNACK message passes (step 4 in Figure 4) unmodified through the client port and is ultimately routed to the client. The client completes the three-way handshake by acknowledging the SYN from the CPU. Typically, the client piggybacks the ACK with data which in this case is the HTTP GET request containing the URL. The ACK and data from the client is again trapped (step 5 in Figure 4) by classifier CC1 (in Table I) at the client port and is forwarded to the CPU. The CPU receives the ACK and data (step 6 in Figure 4) from the client. This completes the three-way handshake at the CPU. The CPU then parses the HTTP GET request data to check if it has received the complete URL. If it has not received the complete URL, it waits for more client data (not shown in the figure) to make its routing decision.

Once the complete URL has been received and the routing decision has been made, the CPU initiates a second TCP connection to the appropriate cluster node (say S1) by sending (step 7 in Figure 4) a SYN packet to it. This time the CPU masquerades as the client and uses the client's IP address (CA) and port (CP) as the source address and the port number, respectively. Recall that all cluster nodes besides being aliased to VIP, also have their own unique IP and MAC addresses. The SYN packet is forwarded to the chosen cluster node using its MAC address. The CPU also sets the starting sequence number to CSEQ, which is the same sequence number that the client initially chose. This way the acknowledgments that are sent back from the server to the client do not need any sequence number translations in the switch.

The SYN packet from the CPU travels through (step 8 in Figure 4) the server port unmodified. On receipt of the SYN, the server sends a SYNACK message addressed to the client. The server independently chooses a starting sequence number, say SSEQ. The SYNACK packet from the server is trapped (step 9 in Figure 4) at the server port by classifier CS2 (in Table II). The server port controller snoops the initial sequence number chosen by the server, *viz* SSEQ as well as the client address CA and port number CP from the SYNACK. It then installs temporary classifiers CS3 and CS4 (in Table II). Classifier CS3 traps all client generated packets and translates the sequence numbers of the ACKs from a base of DSEQ to SSEQ.Classifier

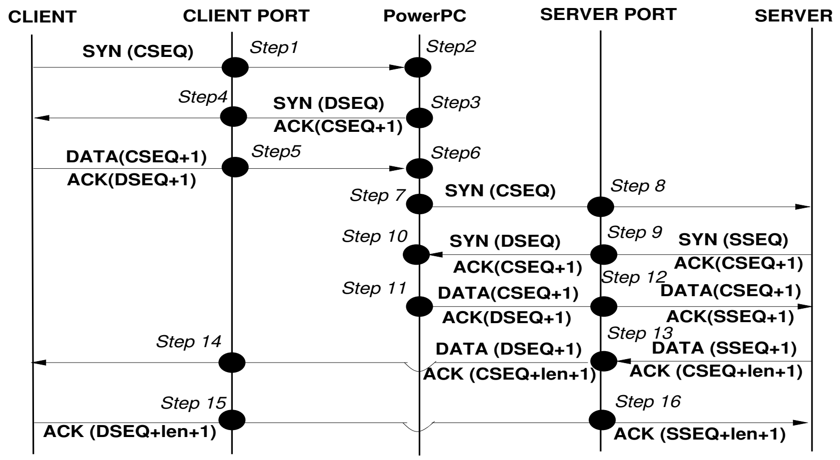


Fig. 4. Flow diagram indicating the steps involved in switching based on Layer 5 information

Client Port Classifiers for HTTP				
ID	DA:DP:SA:SP:Flg	Type	Pri	Action
CC1	VIP:80:*:*:ANY	Perm	P_1	Fwd to CPU
CC2	VIP:80:CA:CP:ANY	Temp	P_2	Fwd to S1

TABLE I
CLASSIFIERS AT THE CLIENT PORT CONTROLLER.

CS4 traps all packets originating at the server and belonging to this connection and translates the sequence number of the data from a base of SSEQ to DSEQ. These classifieres set the stage for the ensuing splicing. Note that DSEQ can be computed locally since all port controllers know the hash function H. After the sequence number translation, the SYNACK is forwarded to the CPU.

The CPU receives the SYNACK (step 10 in Figure 4). It then forwards (step 11 in Figure 4) the acknowledged client data stored at the CPU to the server. This may involve multiple exchanges (not shown in the figure) between the server and the CPU. Data packets sent to the server by the CPU undergo header translation (step 12 in Figure 4) at the server port controller. Once all acknowledged client data has been successfully received by the server, the CPU can get itself out of the data path by splicing the connections [1].

Splicing is accomplished by installing temporary classifier CC2 at the client port controller. Classifier CC2 (in Table I) overrides the default permanent classifier CC1 and forwards all packets belonging to this specific connection directly to the server node instead of forwarding it to the CPU. Also, classifier CS4 (in Table II) at the server port is updated so that packets belonging to this connection and destined to the client are directly forwarded to the client instead of the CPU. After splicing, packets from the server arriving at the server port and matching the classifier CS4 (in Table II) are sent directly to the client after sequence number translation. The client port does not perform any transformation on the packets destined to the client. ACKs from the client are now trapped by classifier CC2 (in Table I) at the client port and are directly forwarded to the server port for delivery to the server node. At the server node, the ACKs match classifier CS4 (in Table II) and undergo sequence number translation and are then delivered to the server. The temporary classifiers are timed out after configurable periods of inactivity. The search engine is equipped with hardware mechanisms to identify inactive classification entries and automatically add them to a list of inactive classifiers. A low priority task processes this list and takes appropriate action.

B. Processing at CPU

Although the bulk of the layer 4 processing takes place at the port controllers, the CPU also plays an important role. It acts as the end-points for the TCP connections to the client and the server, copies data between the connections until they are spliced, and finally splices the connection by sending the appropriate control messages to the port controllers. There are a few subtle issues that need to be dealt with when splicing TCP connections. One of them, *viz* the handling of TCP options [16] deserves special attention. TCP supports various optional functions such as maximum segment size (MSS), window scale factor, timestamp, and selective acknowledgment (SACK) [12]. These options are negotiated during the initial TCP handshake and the set of options supported by both the client and the server are used for the connection. Since the L5 system proxies on behalf of the server, it has to negotiate TCP options with the client. However, at the time of connection setup, the L5 system does not know the specific server node the connection will be routed to and the capabilities of the TCP stack in that server node.

[1]The splicing of the connections may also occur at a later point depending on the layer 5 protocol processor.

As a result, option negotiation becomes a bit tricky. If the switch accepts a specific option which is not supported by the server node to which the connection is ultimately routed to, splicing the connections becomes impossible. The easiest way for the L5 system to handle this problem is to reject all TCP options. A better and preferred approach is to query all the servers in the cluster and to enumerate the minimum set of options supported by all nodes in the cluster. It can then support this minimal set of options while setting up connections with the clients.

For most TCP options, the port controllers do not have to perform any complicated operations. The only exception is SACK. With SACK, the receiver can inform the sender about all non-contiguous segments that have arrived successfully. This is represented as a list of blocks of contiguous sequence space identified by the first and last sequence numbers of the block. This essentially means that the server port controller may have to perform multiple sequence number translations in packets that contain the SACK option. Alternatively, packets with SACK option can be be treated as exception packets that are handled by the CPU.

C. Performance

As mentioned before, the port controllers can perform layer 2-3-4 processing at wire-speed and are not the bottleneck. It is the performance of the processor complex that limits the throughput of the L5 system. We measured the overheads associated with different steps of the datapath executed at the CPU. The measurements were taken on a processor complex equipped with a 233 MHz PowerPC 603e processor (the same used in the switch), 32 KB of data and instruction caches, 512 MB of memory running OS Open. We instrumented the datapath for detailed profiling of various processing steps shown in Figure 4. The instrumented datapath was used to capture a sequential flow of time-stamped events. The time-stamps are of sub-microsecond granularity and are taken by reading a real-time clock which is an integral part of the PowerPC CPUs.

Our measurements show that the overhead associated with steps 2 & 3 (Figure 4) combined is 42 μsecs. The overhead associated with step 6 is 13 μsecs. Time taken for step 7 is 52 μsecs and Step 10 & 11 together take 22 μsecs. Hence the total overhead of layer 4 processing at the CPU is 129 μsecs. Our results show that the L5 system is capable of handling over 7000 layer 5 sessions per second. Assuming an average transfer size of 15 KB [2] per session, this should be able to sustain the throughput of a Gigabit link.

Clearly, in addition to layer 4 processing the CPU also has to handle layer 5 datapath functions. The overhead of these operations may affect the total throughput of the L5 system. In later sections we will discuss the cost of two cases of layer 5 processing and show that it is small compared to the layer 4 data handling.

[2] 15 KB is the average transfer size for the SPECweb96 [17] benchmark

Server Port Classifiers for HTTP				
ID	DA:DP:SA:SP:Flg	Type	Pri	Action
CS1	*:*:VIP:80:ANY	Perm	P_1	Fwd to CPU
CS2	*:*:VIP:80:SYN	Perm	P_2	Learn seq# Inst CS3 & CS4 Fwd to CPU
CS3	VIP:80:CA:CP:ACK	Temp	P_3	Trans seq# Fwd to dest
CS4	CA:CP:VIP:80:ANY	Temp	P_3	Trans seq# Fwd to CPU/CA

TABLE II
CLASSIFIERS AT THE SERVER PORT CONTROLLER.

IV. HTTP ROUTER

In this section, we explore the use of the L5 system in routing HTTP requests using URLs and other session level information. This is particularly useful in environments where the L5 system is used as a front end to a cluster of Web caches and/or Web servers. As a front end to a cluster of Web caches, the L5 system ensures that there is cache affinity when it is making the decision to route a HTTP request. This effectively increases the number of pages that are cached in the cluster resulting in a greater hit rate. Content-based dispatching can also be used as an alternative to sharing a distributed file system across a cluster of web servers. In general, the requirements for each of these environments are slightly different and it is worthwhile to consider them separately.

A. Dispatching to Web Caches

Recently, the need for improved Web performance has resulted in the creation of distributed Web caches that cooperate with each other to increase the amount of web coverage that they can provide. Some of these networks of caches are arranged in a hierarchical fashion [6], [19]. Web caches can also be organized as a loose cluster of distributed caches. In this case, each cache keeps track of the pages that are cached at their peers by running special cache coherency protocols like the Inter-Cache Protocol (ICP) [20]. Alternatively the Cache Array Routing Protocol (CARP) [18], can be used to direct the request to the appropriate cache by using an implicit mapping between the URLs and the caches.

An L5 system working as an HTTP router obviates the need for caches to run ICP and also reduces unnecessary inter-cache transfers of web pages. Additionally an L5 system can easily identify the non-cacheable pages, such as CGI scripts, and forward these requests directly to the appropriate server. This not only reduces transfer latencies, but lessens the load on the caches. The L5 system also improves the effective throughput of the cache cluster by partitioning the content among the caches. Content partitioning reduces the working set size of the content at each node and thus improves the likelihood that the pages are cached in memory rather than stored on the disk.

B. Dispatching to Web Servers

Content-based dispatching can be quite useful in the management of large Web sites, which host millions of Web pages. As mentioned before, these sites use multiple Web servers organized as a cluster to handle high volumes of traffic. The server cluster is front-ended by a layer 4 load balancer to distribute traffic among the server nodes. Since layer 4 dispatching is content blind, all content has to be accessible from each node in the cluster. As was discussed in the introduction, both content duplication and a shared file system are not satisfactory solutions, while content un-aware load distribution may result in significant load variation between the servers of the system.

The L5 system working as a HTTP router is a perfect solution for this environment. By dispatching HTTP requests based on URLs, the L5 system obviates the need for a distributed file system or a complete replication of the content. It allows the content to be partitioned or partially replicated based on performance needs, resulting in significant improvement in the efficiency of the server cluster.

In order to quantify the impact of layer 5 switching on server performance, we conducted a simple experiment using a small cluster of Web servers. Our testbed consisted of three IBM RS/6000 model 43P-200 servers running AIX 4.2, with eight PCs working as clients. The servers were equipped with a PowerPC 604e CPU running at 200 MHz with 32 KB of on-chip 4-way associative instruction and data caches, a 512 KB direct mapped secondary cache, and 128 MB of RAM. The client machines were 266 MHz Pentium II PCs running Linux 2.0.35. Each of the servers was running the Apache version 1.2.4 Web server.

We used the SPECweb96 [17] benchmark to generate client workload for our server cluster. The workload generated by SPECweb is designed to mimic the workload on regular Web servers. More specifically, the workload mix is built out of files in four classes: files less than 1KB account for 35% of all requests, files between 1KB and 10KB account for 50% of requests, 14% between 10KB and 100KB, and finally 1% between 100KB and 1MB. There are 9 discrete sizes within each class (e.g. 1 KB, 2 KB, on up to 9KB, then 10 KB, 20 KB, through 90KB, etc.), resulting in a total of 36 different files in each directory (9 in each of the 4 classes). The number of directories is based on the target workload. For example, with a workload of 500 requests per second there were a total of 100 directories. Accesses within a class are not evenly distributed; they are allocated using a Poisson distribution centered around the midpoint within the class. The resulting access pattern mimics the behavior where some files (such as "index.html") are more popular than the rest, and some files (such as "mydog.gif") are rarely requested.

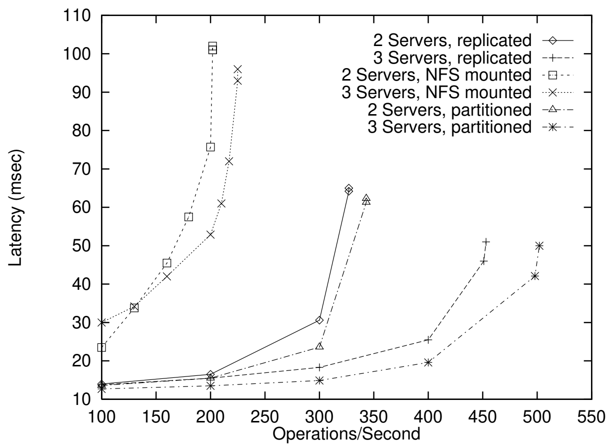


Fig. 5. SPECWeb96 performance on a cluster with 2/3 servers

We conducted three sets of experiments - one where the entire set of files was replicated on each of the servers, a second set with some of the files shared using NFS, and a third set with a partitioned fileset and content aware dispatching. For the first experiment the entire fileset consisting of 100 directories each with 36 files, was replicated on all the server nodes. For the NFS experiments, we partitioned the fileset equally across the different servers and NFS mounted the remaining filesets onto each of the servers. For instance, with 3 servers in the cluster, each server had a third of the files locally whereas 2/3 of the files were NFS mounted from the other two servers. For the third set of experiments, the fileset was partitioned in the same way as in the NFS experiment. However, in this case they were not mounted on the other nodes, rather the client requests were appropriately routed to the servers which hosted them.

Figure 5 shows the latency vs. throughput plots for the three sets of experiments. As shown in the figure, with completely replicated content, a two server cluster was able to register a SPECWeb96 performance of over 300 ops/sec, and a three server cluster was able to support close to 450 ops/sec. With an NFS mounted file system, the server cluster performed very poorly and barely supported 200 ops/sec with three nodes in the cluster. This is not surprising given that a majority of the requests would not be found locally on the servers. If the file was not cached locally the web server would first have to obtain the file through NFS and then serve it out to the client which effectively doubled the work involved. The highest throughput was achieved in the final set of experiments where the client requests were appropriately routed to the servers. In this case the cluster of 3 servers was able to support close to 500 ops/sec. It is interesting to note that the final experiment achieved a greater throughput than the first one where the content was fully replicated. The reason for this higher performance is because in the last case each server sees a smaller set of distinct requests and so the working set size is reduced. This improves the likelihood of a server being able to serve the request from its memory.

B.1 Content to Server Mapping

To be able to dispatch HTTP requests based on URLs, the L5 system has to know the mapping from the URL to the web server (or cache) on which the page resides. As a front-end to a cluster of Web caches it may be sufficient to route the request based on a simple hash of the URL. A drawback of this approach is that the hash function assigns any given URL to a single Web cache. If there are some "hot" pages that are accessed very frequently then this scheme can result in a rather poor load distribution as most of the requests will be routed to a single cache in the cluster. One way of alleviating this problem is to modify the mapping function so that a URL maps to a set of candidate web caches. The request is then forwarded to the least loaded web cache. Alternatively, a mapping from the URL to Web cache can be built on the fly as the initial HTTP requests are dispatched [13]. When a new request arrives, it is assigned to the least loaded cache. A hash table is maintained that maps the requested URL to the cluster node it is routed to. When a repeat request arrives, a simple hash lookup identifies the cluster node on which the page can be found. This scheme can be modified to allow a mapping between a request and a set of nodes, with an incoming request being assigned to the least loaded cache in this set [13].

When the L5 system is used as a front-end to a cluster of Web servers the problem is slightly more complicated. In this case, the L5 system is not responsible for distributing the content to different server nodes and hence cannot learn it automatically. Consequently, its first task is to identify the location of the content. If the Web pages are organized in a structured fashion, a simple static configuration may be sufficient. We are implementing a more ambitious scheme where the L5 system learns the mapping using an URL Resolution Protocol (URP) [1].

B.2 URL Lookup

One of the important components of HTTP routing is finding the server node that hosts the requested Web page. The data structure used to store this mapping of URLs to server nodes, depends on the specific application environment. For example, for routing in a Web cache cluster where the content space has very little structure, a hash table is probably the best choice. For routing in a Web server cluster, it may be possible to exploit the structure in the content space by using a data structure that allows prefix matches. We are considering both hash tables and data structures that facilitate prefix matching for use in the L5 system. For prefix matching we use multilevel hash trees where each level in the hash tree corresponds to a level in the URL.

Maxlevel	Lookup time (μsec)	Memory (KB)
1	2.38	8.4
2	4.69	30.10
3	8.98	116.93
4	9.93	416.55
5	10.00	462.26

TABLE III
URL LOOKUP PERFORMANCE USING HASH TREE.

To estimate the overhead of URL lookup, we constructed a multilevel hash tree with about 20,000 URLs from the 1996 Olympic Web site. We started with a simple hash function and set the default size of all hash buckets to 256. After populating the tree with all the URLs, we examined the hash buckets and revised the hash functions and bucket sizes to minimize overflow and underflow conditions. Table III shows the overhead of URL lookup both in terms of the lookup time as well as the memory consumption for different levels of aggregation. Since we don't know exactly how the pages were distributed across the different servers we can't predict the exact amount of aggregation that is attainable. Rather we list the results for different levels of aggregation where an aggregation to level 3 implies that only the first three components of the path name are required to identify the server (or servers) on which the page resides. The lookup time was computed by measuring the average search time for the URLs over a trace of 7.5 million requests served by the 1996 Olympic Web server.

From Table III it is clear that aggregation dramatically reduces memory consumption. To compare this result to the performance of the flat hash scheme, we created a flat hash table with the number of buckets equal to the total buckets in the multilevel hash tree. The average lookup time using the flat hash scheme was 5.87 μsec. The amount of memory required to store the flat hash table was 1 MB – more than double the amount required for the full multilevel hash tree. This is due to the fact that in the flat hash table each node stores the full URL as opposed to the partial URLs stored by each node in the multilevel hash tree.

C. Performance of Content to Server Mapping

The cost of performing this mapping, i.e., parsing the URL and finding the server that the request should be directed to, can affect the performance of the L5 system. We evaluate this cost by experimentation. Due to the large number of clients and servers required to saturate the system, it is very difficult to measure the actual throughput of the L5 system. Hence, we estimate the throughput of the L5 system as a HTTP router by measuring the overhead of HTTP parsing and URL lookup.

Our results show that the overhead of HTTP parsing and URL lookup is substantially smaller than the layer 4 functions performed by the CPU. On average the combined cost of HTTP parsing and URL lookup is about 15 μsec. So it takes a total of 144 μsec (129 μsec due to layer 4 processing) to route a HTTP request. This translates to a throughput of around 7000 connections per second.

V. Self Learning SSL Dispatcher

Electronic commerce (e-commerce) applications are one of the fastest growing segments of the Internet. The most conspicuous feature that differentiates an e-commerce application from other Internet applications is security. In almost all instances, the Secure Sockets Layer (SSL) protocol [8] is used to ensure security in e-commerce applications. While the importance of SSL in the context of e-commerce applications cannot be overstressed, it adds significant overhead to protocol processing, especially at the server end. Consequently, large e-commerce installations use clusters of servers to improve scalability. In this section, we discuss the problems associated with dispatching SSL sessions to the nodes in a server cluster and show how the L5 system can be used to address this problem.

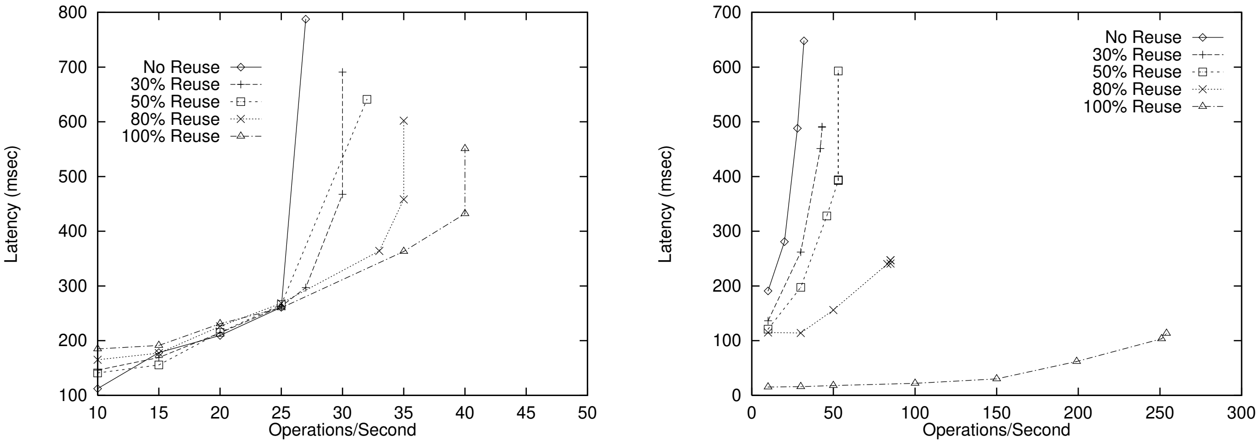
A. SSL Session Reuse

SSL typically runs over TCP. A client wishing to establish a secure channel to the server has to first setup a TCP connection to the server. Once the TCP connection is established, the client and the server authenticate each other and exchange session keys. This phase is known as the SSL handshake and is computationally very expensive as it typically involves public key cryptography. Once the handshake is complete, the two parties share a secret which is used to construct a secure channel between the client and the server. In contrast to the handshake performed during the establishment of a new session, the reestablishment of an SSL session is relatively simple. The client specifies the session ID of the old or existing session it wishes to reuse. The server checks to determine if the state associated with this session is still in its cache. If the session state exists in the cache, it uses the stored secret to create keys for the secure channel. The latency involved in setting up a secure connection using cached session state is an order of magnitude lower than a full SSL handshake [2].

In this paper we focus on SSL session reuse in the context of a server cluster. To better understand the problem consider a scenario similar to the one depicted in Figure 3 where a cluster of Web servers are serving HTTP requests over SSL. The L5 system is responsible for dispatching the incoming SSL connections to the server nodes with the objective of balancing the load among them. This scenario is typical of many large e-commerce installations which have to handle thousands of secure Web transactions every second. At present, the common practice is to use a layer 4 load balancing switch that distributes connections to server nodes disregarding SSL session level information. Since the server nodes do not share their session caches, this approach leads to poor SSL session reuse efficiency. An easy way to improve the reuse efficiency is to route all connections from a client to the same server node. Unfortunately, this approach may cause severe load imbalance among the nodes in the cluster. A layer 4 dispatcher cannot distinguish between two different clients that are behind the same firewall or proxy. As a result, it routes connections originating from all clients behind a firewall or a proxy to the same server node, leading to massive load imbalance. Since a large percentage of Internet clients are behind proxies and firewalls, this poses a serious problem with no obvious solutions.

From the above discussion, it is clear that a cluster environment complicates session reuse unless the cluster nodes share the session cache. While sharing of session cache is feasible, there are many technical obstacles that makes it difficult. First, for security reasons, it is not advisable to make the session cache accessible over the network. Even if one disregards the security advisory, at a minimum one has to make sure that both the session caches and their clients authenticate each other appropriately. Creating such an infrastructure requires a complex configuration and is an administrative nightmare. Second, this approach requires modifications to the SSL libraries and standardization of session cache interfaces so that different implementations of SSL can share the session state information with each other.

An elegant and much better alternative to sharing the SSL session cache among the nodes in a cluster is to use a SSL session aware dispatcher. Such a dispatcher can learn the SSL session to cluster node mappings by snooping on SSL messages and can dispatch the session reuse requests to the appropriate server nodes using this mapping. In the following, we describe how the L5 system can be used for this purpose.



(a) Apache cluster with SSL unaware dispatching. (b) Apache cluster with SSL aware dispatching.

Fig. 6. Impact of session session aware dispatching on server performance.

B. *Session Aware Dispatching*

The basic steps involved in session aware dispatching of SSL connections are similar to the URL based routing discussed in the previous section. As the client request to setup a TCP connection arrives at the L5 system, it is intercepted by the port controller and is forwarded to the CPU. The CPU establishes a connection to the client and waits for the SSL session setup message. Upon receiving the SSL session setup message, the SSL protocol processor parses the message and extracts the session ID contained in the message. Based on the session ID it decides which server node has session state corresponding to this session. Once the appropriate server is identified, it sets up a second connection to the server node and forwards the setup request. The response from the server is intercepted by the port controller and is forwarded to the CPU. The CPU parses the message and extracts the session ID information contained in it to update its session ID to server node mapping. It then splices the two connections and gets out of the data path.

Understanding the steps involved in SSL session aware dispatching requires a knowledge of the message flow involved in SSL session setup. After the underlying TCP session has been established, the client initiates the SSL connection by sending a *Hello* message to the server. The Hello message includes a session ID field which is empty if a new SSL session is to be established. In response to the client Hello, the server picks a session ID and then sends a Hello of its own which includes the session ID. The server Hello is followed by the server certificate which contains the server's public key. The client verifies the certificate, generates a secret, and encrypts it with the public key obtained from the server's certificate. This is sent to the server which performs a decryption using its private key, thus obtaining the secret. This shared secret is then used to generate symmetric keys for encryption and message authentication. Until this point all the messages are exchanged in the clear and are potentially available to the L5 system. Once the secret keys have been generated by both sides, the client and the server start using encryption and message authentication.

When reconnecting to the same server, a client can reuse the session state established during a previous handshake. In this case the client sends a Hello message which includes the session ID of the session it wishes to reuse. If the server still has the session state in its cache, the client and the server undergo a short exchange, leading to the reestablishment of the session state. Otherwise, the server picks a new session ID and a full handshake is performed.

Figure 6 shows the impact of session aware dispatching on the Apache server 1.2.4 using SSLeay 0.8. For this experiment, we used the three identical IBM RS/6000 servers that were described in Section IV. A cluster of PCs running SPECweb96, suitably modified to generate HTTPS [3] traffic, were used as clients. We used RC4 [15] for data encryption and MD5 [14] for message authentication, since these are the most commonly used ciphers in real life [4]. We varied the degree of session reuse from 0-100%. When session reuse is 0% all SSL sessions setup between the server and the clients require a full handshake. When session reuse is 100%, only the first SSL session setup between the server and a client involves a full handshake. All subsequent connections reuse the already established session state between the server and the client. When the percentage of session reuse is between 0 and 100, the clients reuse the same session for a certain number of times depending on the value of the reuse percentage.

Figure 6(a) shows the performance of the server cluster when the load balancer is unaware of layer 5 sessions and dispatches connections based on layer 4 information only. From the figure it is clear that when SSL sessions are

[3] HTTP requests over SSL.

[4] Both Netscape and Microsoft Web browsers use RC4 and MD5 as defaults.

blindly dispatched to nodes in the cluster, the aggregate throughput of the cluster saturates at around 30-35 connections per second depending on the degree of session reuse. As expected, the degree of session reuse has little impact on performance. There is however an interesting anomaly that can be observed at low utilizations where the latency increases with the degree of session reuse. This is due to the fact that Apache maintains a global cache to store all SSL session state in addition to the per-process cache maintained by the server processes. While processing a reuse request, the server process first checks its local cache for a hit. If it fails to find a match in its local cache it searches the global cache for a hit. As the degree of session reuse increases, so does this futile search through the global cache. This results in an increased latency for the connections that request a reuse of session state. This results in a higher average latency as the level of session reuse is increased. When the utilization level is sufficiently high a significant amount of time is spent waiting for the CPU and so this effect is masked at higher loads.

Figure 6(b) demonstrates how SSL session aware dispatching can substantially improve the performance of the server cluster. In this case as the degree of reuse increases so does the throughput of the server cluster. With 80% session reuse the three server cluster can sustain a throughput of about 100 connections per second, almost triple the throughput achieved in the previous experiment at the same level of reuse. With 100% reuse we observe a six fold improvement in the performance.

We are in the process of evaluating the performance of the L5 system as a session aware dispatcher of SSL connections. Preliminary measurements indicate that the CPU overhead for routing SSL sessions based on session ID is very close to that of routing HTTP sessions using URLs.

VI. Conclusions

In this paper, we share our experiences in the design and implementation of a content-based switching system named L5. The L5 system uses application level information in addition to layer 2-3-4 information to route traffic in the network. It combines the functionalities of an application layer proxy and the data handling capabilities of a switch into a single system. By migrating all of the layer 4 processing to the port-controllers we are able to achieve a high throughput through the L5 system. In this paper, we also discuss the usefulness of the L5 system as a content based router front-ending a server cluster. Using a popular web benchmark, we clearly demonstrate the benefits of the L5 system in the context of two application examples, namely URL based routing and session aware dispatching of SSL connections.

The work presented in this paper can be extended in many ways. We are currently exploring the use of the L5 system for content based service differentiation. Content based service differentiation is particularly useful in large e-commerce sites to differentiate serious buyers from casual browsers. Content based service differentiation can also be used to provide service differentiation based on user profiles. Web servers often set cookies to identify users and track session information. The L5 system can make use of the cookies in the HTTP requests to determine the level of service required by a given connection. We are also investigating the usefulness of the L5 system as a content based filter at ISP and corporate access gateways.

References

[1] G. Apostolopoulos, V. Peris, P. Pradhan, and D. Saha. L5: A self learning layer 5 switch. Technical Report RC21461, IBM, T.J. Watson Research Center, 1999.

[2] G. Apostolopoulos, V. Peris, and D. Saha. Transport layer security: How much does it really cost? In *Proceedings of the IEEE INFOCOM*, 1999.

[3] Web Server Farm Performance. White Paper, Arrowpoint Communications, 1998.

[4] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. IETF Request for Comments 1945, October 1995.

[5] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform Resource Locators (URL). IETF Request for Comments 1738, December 1994.

[6] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996 USENIX Technical Conference*, Januaray 1996.

[7] Cisco Local Director. Technical White Paper, 1998. Cisco Systems.

[8] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 Protocol. Netscape Communications Corporation, November 1996.

[9] ISI for DARPA. Transport Control Protocol. RFC 793, September 1981.

[10] M. Leech and D. Koblas. SOCKS Protocol Version 5. IETF Request for Comments 1928, April 1996.

[11] D. Maltz and P. Bhagwat. Application Layer Proxy Performance Using TCP Splice. IBM Technical Report RC-21139, March 1998.

[12] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. IETF Request for Comments, 2018, October 1996.

[13] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality Aware Request Distribution in Cluster-based Network Servers. In *Architectural Support for Programming Languages and Operating Systems*, 1998.

[14] R. Rivest. The MD5 Message Digest Algorithm. IETF Request for Comments 1321, April 1992.

[15] B. Schneier. *Applied Cryptography*. Wiley, New York, 1996.

[16] W. R. Stevens. *TCP/IP Illustrated Volume 1*. Addison-Wesley, New York, 1996.

[17] The Standard Performance Evaluation Corporation. Specweb96, 1996. http://www.spec.org/osg/web96.

[18] V. Vallopilli and K. Ross. Cache Array Routing Protocol Version 1.0. Internet Draft draft-vinod-carp-v1-03.txt, February 1998.

[19] D. Wessels. Squid internet cache object. URL: `http://squid.nlanr.net/Squid/`, 1998.

[20] D. Wessels and K. Claffy. Internet Cache Protocol Version 2. IETF Request for Comments 2186, September 1997.