Scalability of content-aware server switches for cluster-based Web information systems

Mauro Andreolini
Department of Information, Systems and Production
University of Tor Vergata
Roma, Italy 00133
andreolini@ing.uniroma2.it

Michele Colajanni
Department of Information Engineering
University of Modena
Modena, Italy 41100
colajanni@unimo.it

Marcello Nuccio
Department of Information Engineering
University of Modena
Modena, Italy 41100
marcenuc@weblab.ing.unimo.it

ABSTRACT

A cluster-based architecture with a front-end Web switch and locally distributed servers seems the most appreciated solution to face the ever increasing demand for complex services offered through Web interfaces. The complexity of the novel services is often related to the possibility of content-level identification and personalization that can be achieved through a content-aware front-end component. It is common belief that content-based operations prevent the scalability of the Web cluster, to the extent that a content-aware switch alone is seldom used as the front-end of a popular Web site. In this paper, we demonstrate that a careful design and optimized implementation choices based on a modern PC-based architecture can give a Web switch with content-aware functionality and very limited overheads. We present the design and prototype implementation of a so called one-way system based on Linux kernel, single CPU and SMP-based architectures, for HTTP/1.0 and HTTP/1.1 protocols. The experimental results confirm that the proposed solution is extremely scalable, thus making a content-aware Web switch a viable solution to the performance requirements of the majority of cluster-based architectures.

Keywords

Cluster-based systems, Network servers, Scalability, Performance evaluation, Content-aware routing

1. INTRODUCTION

Scalability remains a main requirement of a modern Web-based system that should be able to accommodate for user requests that augment in number and complexity. Unfortunately, upgrading just the number of servers does not represent a valid solution to the scalability problem, because this would move the bottleneck from the (back-end) server side to the front-end side. This risk is even more serious when we consider that new Web-based services require that the front-end component can catch from a request the largest set of information that exists at application level but not at TCP level. Content-aware features augment the front-end scalability issues of one-two orders of magnitude. Many solutions have appeared to improve the delivery of Web content [6, 4, 9, 12] through *locally distributed Web-server systems*, briefly *Web clusters*. For a recent survey on the topic, see [7].

Basically, a Web cluster is a set of server machines that are interconnected through a high-speed LAN. The cluster is publicized through one site name and one *virtual IP address* that typically corresponds to the address of a dedicated front-end node. This important component, also called *Web switch*, is the main focus of this paper. It acts as an interface between the nodes of the cluster and the rest of the Internet, thus masking the distributed architecture of the site to the users and the clients. The Web switch receives all client requests and routes them to a Web server node through some centralized *dispatching policy*. We distinguish *layer-4* from *layer-7* Web switches. A layer-4 switch performs content-blind routing that is, it does not take into account any content information in the client request in performing assigning decisions. On the other hand, a layer-7 Web switch performs content-aware routing: it first establishes a complete TCP connection with clients, parses each request and assigns a Web server node according to the content. Content-aware routing allows a Web cluster to use sophisticated dispatching strategies, improves cache hit rates, permits content partitioning and gets a much larger set of user/client information. However, it tends not to be used as a front-end component of a popular Web-based information system because it has been demonstrated to be less efficient than a layer-4 Web switch. As an example, Aron et al. [4] show that the peak throughput achieved by a layer-7 switch is limited to 3500 conn/sec, while a software based layer-4 switch implemented on the same hardware is able to sustain a throughput up to 20000 conn/sec. To improve scalability of layer-7 architectures, alternative solutions for scalable Web-server systems, which combine content-blind and content-aware request functionality, have been proposed, e.g. [18, 27].

The motivation of this paper comes from the observation that the absolute efficiency is not the right measure to judge the possibility of using a content-aware Web switch. Indeed, its performance should be related to the operational requirements that a Web switch should satisfy in a realistic multi-tier environment. This includes the inter-connection of the Web cluster to the Internet (for example, the large majority of Web clusters for economic reasons does not use more than T3-based connections, that have a peak bandwidth of 45 Mbps), the HTTP servers (with typical workload and modern hardware, they are not the system bottleneck anymore, unless they have to manage secure transmissions), and the back-end servers (that can easily become the system bottleneck, when the dynamic requests are computationally expensive). Moreover, when the classes of services provided by the Web site require peak throughputs higher than 40-50 Mbps, it is more likely that a different architecture should be considered, for example a system distributed over a geographical area.

These motivations induced us to investigate whether the previous prejudices against layer-7 Web switches are still valid when one

WWW2003, May 20–24, 2003, Budapest, Hungary.
ISBN 963-311-355-5.

considers modern hardware and multi-tier architectures for content-aware distribution in cluster-based Web information systems.

We describe the design and implementation of an efficient, content-aware Web switch that takes advantage of all possible features and optimizations of modern PC-based architecture. We demonstrate that careful design and implementation choices produce a Web switch with content-aware functionalities and very limited overheads. A careful analysis of its performance demonstrates that the proposed solution is extremely scalable, thus making a content-aware Web switch a viable solution to the performance requirements of the majority of popular Web sites based on cluster architectures. The most important contributions of the layer-7 Web switch are outlined below and discussed in the following sections.

- Almost all Web switches are based on two-way architectures that, even if implemented at the kernel level, are less efficient because both requests and responses transit through the Web switch [16, 21, 8, 19, 30, 10, 15, 13, 24, 29, 2, 28, 5]. On the other hand, as in [4], the proposed Web switch uses a one-way architecture where just the client-to-server requests flow through the Web switch, while the larger server-to-client responses use another way.
- All content-aware dispatching features are implemented and integrated at the kernel level of a Linux operating system.
- The design and implementation avoid the most serious inefficiencies existing in other known implementations.
- The Web switch can operate on a single processor architecture or even SMP architectures (most experiments refer to a dual Pentium architecture). In particular, we exploited the *spinlock* primitives [26] to guarantee the most efficient mutual access to different CPUs to data structures.
- Transfers of TCP connections between the Web switch and the server are based on the so called *TCP Handoff* protocol that has been proposed by Aron et al. for the FreeBSD Unix [4]. Our version is the first that has been specifically designed and implemented for Linux operating systems. (The FreeBSD and Linux kernels are so different in the choices about the network-based operations that very few ideas could be taken from the previous TCP Handoff implementation, not to say about the optimizations that work only for Linux-based systems.)
- The Web switch design is highly modular from the point of view of request dispatching policies: we have experimented content-blind, content-aware, server load-aware, and combinations of content- and server load-aware dispatching policies, even if a subset of results can be reported in the paper.
- The content-aware distribution mechanism has been designed to be compliant with both the HTTP/1.0 and HTTP/1.1 protocols.
- The Web servers do not need specific configurations to communicate with the switch node. Hence, it would be possible to change the switch node role without reconfigurations of the entire cluster. This augments the availability of the system.

The implemented Web switch has been subject to a large variety of performance tests. All results confirm that the proposed layer-7 Web switch has a low overhead, even when the Web servers tend to be saturated. Moreover, we show that the Web switch scales pretty well across multiple server nodes. Finally, we also evaluate the performance of the Web cluster under realistic workload conditions. Again, we show that the switch is able to handle several thousands of connections per second without being the bottleneck of the whole system. We can conclude that the proposed one-way architecture is extremely scalable, thus making content-aware routing a viable solution to the requirements of the majority of network services provided by cluster-based architectures.

The rest of this paper is organized as following. In Section 2, we describe main requirements, major issues and our solutions for an efficient design of the layer-7 one-way Web switch. Section 3 outlines two content-aware dispatching policies that we use for the experiments. Section 4 presents the implementation details, with major focuses on the techniques to obtain the best performance from single- and dual-based processor architectures. Section 5 contains the performance study. Section 6 concludes the paper with some final remarks.

2. DESIGN OF A SCALABLE LAYER-7 WEB SWITCH

This section provides a detailed design of the Web switch architecture. We identify the main requirements that any efficient kernel-level implementation should satisfy. We also point out several design choices which lead to the optimization of core components.

2.1 One-way architecture

The first important design choice concerns the flow of the packet traffic to and from the Web cluster. All client requests reach the Web switch, so the most important difference is how the response go to the clients. In so called *two-way* architectures, server responses are directed towards the Web switch, which directs them to clients. In *one-way* architectures, responses are sent by the servers directly to the clients, thus bypassing the Web switch. One-way architectures limit the risks of system bottleneck at the Web switch due to forward and backward handling of each packet. The problem is that a layer-7 one-way architecture is much harder to be implemented than a two-way switch because the distributed architecture must remain transparent to both the user and the client application. Hence, each Web server must be able to change the response packets in such a way that they seem coming from the Web switch which is the only official interface for the clients.

Our one-way solution is based on the TCP Handoff mechanism that has been implemented on Linux operating system. We point out the problems we had in designing efficient core components of the TCP Handoff mechanisms, and give also some hints on how to address issues in SMP practice. To describe the operations performed by the nodes of a Web cluster, we detail the sequence of events activated by a client request. To this purpose, we refer to the time diagram in Figure 1.

A client process connects to the Web switch through the standard TCP/IP protocol. The switch tries to establish a TCP connection with the client through a three-way handshake. Next, the client sends a request. The switch parses the request and extracts the application-level information, such as URL, cookies or SSL identifiers. Next, the switch chooses a Web server among those which are able to serve the requested content. The switch then transfers the TCP connection to the chosen Web server, along with the client request. The Web server re-creates the TCP connection in the same state it was before being transferred. We refer to the *connection state* as to the set of necessary information for cloning the connection on a different node.

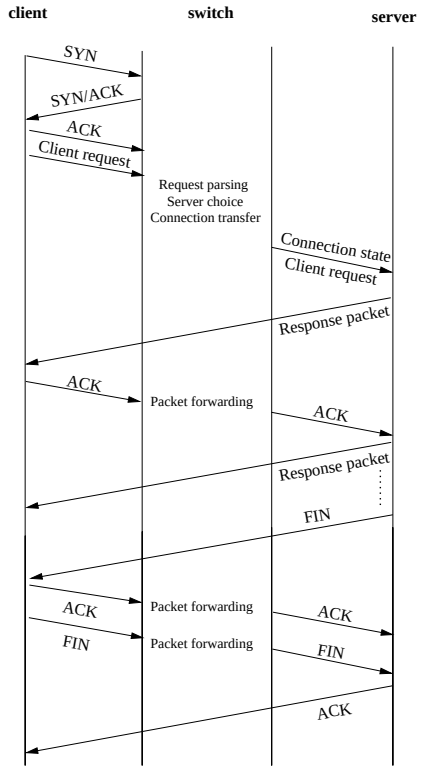

Figure 1: High level view of the TCP Handoff mechanism.

Once the connection has been transferred, the client request is inserted into the server process queues. The server builds a response and sends it directly to the client, thus acting as if it was the Web switch.

The client receives the response packet and sends an acknowledgement (but also future requests, if the HTTP/1.1 or persistent TCP connections are used) to the Web switch. Those packets cannot be sent directly to the server because the client continues to have the impressio that the connection is established with the Web switch. Thus, the switch must keep forwarding client and ACK packets until the TCP connection is closed.

2.2 Efficiency requirements

The TCP connection transfer is one of the most critical phase because it must satisfy a rather complicated set of requirements. We identify also the other main issues for an efficient implementation of the one-way Web cluster.

TCP connection transfer. The whole handoff process requires that established TCP connections be transferred from one node (specifically, the switch) to another (a chosen Web server). To complete this operation, the internal state of a TCP connection, identified by the *Transmission Control Block (TCB)* [23], must be communicated to a Web server node, in order to be able to recreate that connection in the same state it was before being transferred. Besides this, the Web server node must also receive the application-level information (usually, a client request) that started the connection transfer process. Both information (TCP connection state and client request) may be logically chained in an entity which we call *TCP Handoff request*. The name is justified by the following observation: a TCP Handoff request contains the minimum amount of information which is needed to shift the request serving process from one node to another. Upon the receipt of a TCP Handoff request, the Web server extracts the TCB and recreates the TCP connection. Next, it obtains the application-layer information which is tailed into the server process receive queues.

Request parsing. The Web server which must serve a client request is chosen according to (a well-defined portion of) the application-level information contained in the request. Hence, it is necessary to provide a means for parsing the client request, with the goal of extracting the required information which will be used by the dispatching policy. It is worth noting that the parsing functionality is strictly related to the dispatching policy. Different policies may require different portions of a client request, such as parts of an URL, cookies, or even SSL identifiers.

Dispatching mechanism. This is the second most critical phase after the TCP connection transfer. The distribution of client requests across cluster nodes requires an algorithm to take decisions and a mechanism to execute assignment decisions. In this paper, we consider content-aware request scheduling, so the dispatching policy (generally) uses some information extracted by the request parsing mechanism to identify the cluster node which is able to fulfill the request.

Evaluation of Web server load conditions. Many dispatching policies use some information related to the Web server load status to perform a sub-optimal assignment or, at least, to minimize the risks of load unbalance. It has been widely shown that the introduction of state-awareness leads to smarter assignments than those performed by state-blind policies. However, a state-aware algorithm requires a mechanism for evaluating the conditions of a Web server node, which we call *load monitor*. The load monitor may be *centralized* or *distributed*. A centralized monitor runs entirely on the switch and treats the Web server nodes as "black boxes". A distributed monitor collects state information directly on the Web server node, which is then communicated to the switch. The former approach is typically easier to implement, since it does not require a communication mechanism. However, it does not estimate the server load as well as the distributed approach, that is able to access directly to the operating system internals. Server load information tends to become obsolete quickly [11]; thus, it is preferable to avoid time-synchronous updates every n seconds. A better alternative is to update the server load coefficients synchronously with events, like the successful transfer of a connection and the closure of a transferred connection.

Forwarding mechanism. Client packets subsequent to the received response segments cannot be handled anymore by the switch; they must be handled by the appropriate Web server. For this reason, the switch must be enriched with a mechanism for intercepting client packets and for delivering them to the Web server to which the TCP connection has been transferred. This operation requires a modification of some fields of the TCP/IP header, such as the IP and MAC destination addresses which have to be set to those of the real Web server. Such modifications imply the (re)computation of the IP and TCP checksums before the delivery of the packet to the Web server. It is of crucial importance that the forwarding mechanism be as fast as possible, otherwise the traffic between the client and the server is seriously slowed down. Typically, the operations involved in the forwarding module are CPU-bound, because of the TCP and IP checksum computation. Due to the large number of simultaneous open connections handled by a switch, the design of a forwarding mechanism which scales well in an SMP architecture is mandatory.

Transparency. The whole TCP Handoff mechanism must be transparent to the clients. In other words, the responses and the end-to-end semantics of all TCP/IP operations must remain the same as if the client was connected to one Web server. This implies that

response packets must look as if they were sent by the Web switch. Typically, this requires at least the modification of the following fields of a TCP/IP packet: source IP address, and TCP timestamp option [17].

Making the entire mechanism transparent also to the server processes is an important plus, as the Web server applications must be portable and should not require modifications to work in one-way Web clusters.

2.3 Design solutions for efficiency

In this section, we summarize the main design choices that aim to improve the operations of the Web cluster components. Other minor optimizations are possible during the implementation phase, and will be discussed in Section 4.

Cluster architecture. The first important choice for an efficient content-aware distributor concerns the architecture of the cluster. We considered the TCP Handoff mechanism because it is a kernel-based one-way solution, which is intrinsically faster than two-way solutions, and makes up for a better scalability. We paid attention to optimizing most of the software components. To this purpose, we have chosen the Linux operating system kernel (version 2.4.18) as the implementation platform, also because of its efficiency and stability on SMP architectures. In particular, we exploited the *spin-lock* primitives [26] to guarantee mutual access of different CPUs to kernel data structures.

Forwarding mechanism. As already observed, the design of the forwarding mechanism is a crucial issue, as the speed of packet forwarding operations is one of the factors that can limit the performance of the entire cluster architecture. In particular, two operations must be performed really fast: discovery of transferred connections; checksum re-computation.

To quickly determine whether a connection has been transferred or not, we choose to store information about transferred TCP connections in a hash table. Each entry in this table stores IP addresses and TCP ports characterizing a TCP connection. Figure 2 shows the idea for efficient lookup. Basically, the access to the hash table is carried out through a *key*, consisting of the source IP address and the source TCP port. The key is mapped by a *hash function* into the index of a list (better said, a *bucket*) of structures identifying transferred connections. If the hash function spreads uniformly different keys (i.e., different connections) into different buckets, then the accesses are usually carried out into several smaller lists. Hence, the usage of a hash table makes access operations (particularly, lookup operations) much more efficient than alternative implementations based on a single chained list. We add the following solution: each bucket is protected by a lock, so that different CPUs can access the table concurrently without incurring in mutual exclusion problems. The hash function is implemented through the 32-bit XOR operator, since it maps different client port numbers into different buckets (if the source IP is fixed):

$$key = (\text{src IP}) \oplus (\text{src TCP port}). \quad (1)$$

With this design choice, successive requests from the same client are mapped into different buckets and may be handled in parallel on a multi-node switch.

Another critical issue is the re-computation of the IP and TCP checksums when a packet has to be forwarded to a Web server. We use the incremental checksum update described in [25] to avoid the re-computation of both checksums.

Switch detection at runtime. In our design, the server does not need any a-priori configuration of the switch IP address. Indeed, this address is obtained from the Handoff request and is stored as in the socket structure pertaining to the connection. In such a way, the server is not tied to a single switch, but it may handle Handoff requests from multiple switches. Using many switches may increment the scalability and the fault-tolerance of the cluster.

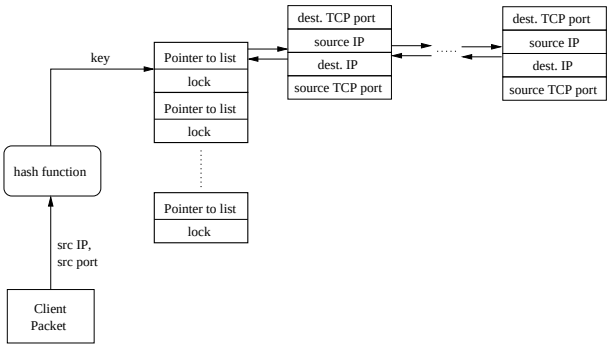

Figure 2: Efficient lookup through a hash table.

3. DISPATCHING POLICIES

In this section, we present the subset of the content-aware dispatching algorithms that are available in our Web switch and are used for the experimental results.

The **Client Aware Policy** [8] is oriented to Web sites providing heterogeneous services with different computational impact on system resources. The set of static and dynamic services provided by the Web site is divided in classes, each one stressing the system components in different ways. The CAP algorithm works as follows. A list of circular pointers to servers is maintained (one for each service class). As soon as a client request is received at the switch, the parser module extracts the embedded URL and identifies the associated service class. Then, a round robin assignment on the given service class is performed, by using the appropriate pointer. The basic observation of CAP is that when the Web site provides heterogeneous services, each client request could stress a different Web system resource. Although the Web switch cannot estimate the service time of a static or dynamic request accurately, it can distinguish the class of the request from the URL and estimate its main impact on each Web system resource. A feasible classification for CAP is to consider disk bound, CPU bound, and network bound services, but other choices are possible depending on the content and services provided by the Web site.

We also implemented the *Locality-Aware Request Distribution* [3, 22], which tends to maximize cache hit rates of static resources. As soon as the Web switch receives an HTTP request, the parser module extracts the URL. Next, it checks whether the requested URL has already been handled by any Web server node. If this is the case, the request is forwarded to that node, unless it is overloaded. To avoid potentially unfair assignments, the server load is estimated through a centralized load monitor that counts the number of active connections for a given request class (static, dynamic). A Web server is considered overloaded if the number of opened connections exceeds a given threshold. If the chosen server is overloaded, the least loaded node is chosen. If the URL has not yet been assigned to a Web server, the least loaded node is chosen as well. The rationale behind LARD is that assigning the same Web object to the same Web server, the requested object is more likely to be found into the disk cache of the server node.

4. IMPLEMENTATION DETAILS

We describe the architectural design and the implementation of

the one-way layer-7 Web switch based on the TCP Handoff approach, called *ClubWeb-1w*. The switch software has been conceived as an extension of the TCP/IP stack under the Linux operating system, kernel version 2.4.18. Particular care has been taken in avoiding slow and unefficient constructs, and in implementing thread-safe code. The implementation takes full advantages of the Linux multi-threaded TCP/IP stack and results particularly efficient on SMP nodes. The cluster operations are implemented through the following software modules: *dispatcher*, *forwarder*, *THOP communication protocol*, *load monitor*. We describe them by showing how they fulfill the design requirements in Section 2.2.

4.1 Dispatcher module

The dispatcher module handles a list of structures representing the pool of Web servers in the cluster. Each structure stores per-server information, such as the destination IP address (or addresses, in case of multiple network interfaces) and a representation of the server load status, that is defined by the dispatching policy. This is necessary, as different dispatching algorithms generally use different parameters for the estimation of server load. The dispatcher module is also responsible for parsing the client request and choosing an appropriate Web server according to a content-aware scheduling policy. In our architecture, it is organized as a set of *hook functions* which can be customized on the basis of the chosen policy. Each hook function handles a specific stage of the dispatching process. We consider the following stages.

Initialization. This hook gives an opportunity for executing operations, such as allocation of system resources, which have to be performed at cluster startup only.

Request parsing. This hook parses the client request and extracts all information which is necessary for the choice of a Web server.

Server selection. This hook takes the extracted client information and, when necessary, the server load status. It has to choose an appropriate server according to some dispatching algorithm. It returns the structure associated to the Web server. The dispatching hook is invoked during the processing of TCP segments containing application-layer requests on a given, configurable, port.

Post request. This hook is called whenever a request has been processed by a Web server and the switch is notified. It may be used to update the server load status in a centralized load monitor.

Shutdown. This hook permits the execution of operations which have to be performed only at cluster shutdown. Typically, these operations release the resources allocated at the cluster startup.

The design through hooks is extremely flexible and modular. The behavior of the switch may be extended and specialized by adding more hooks. Each dispatching policy defines its own hook functions, hence the other modules are unaffected.

4.2 Forwarder module

The forwarder module must intercept client segments belonging to already transferred TCP connections and transmit them to the appropriate Web server. To this purpose, we implemented an efficient hash table of structures representing transferred TCP connections, as described in Section 2.2. The TCP protocol allows port reuse when a connection is in the TIME_WAIT state [23]. Under very bursty traffic conditions, it may happen that a client want to reuse a TCP connection in the TIME_WAIT state, by sending a SYN packet. In this case, the forwarder delivers the SYN directly to the Web server, instead of performing the three-way handshake. As a consequence, the following client request would have no chance of being parsed by the Web switch. To address this issue, we have implemented a further functionality in the forwarder module. We check every client packet for a FIN or a RST flag, which indicates an intention of closing the TCP connection. If a FIN or a RST is intercepted, the matching element in the table of transferred connections is marked as "closed". When the forwarder module receives a subsequent SYN packet from the client, it checks whether the matching element in the table of transferred connections is marked as closed. This event means that a TCP port has been reused. As a consequence, the element is removed from the hash table, to avoid any forwarding towards the previous Web server. Furthermore, the client SYN is passed to the upper layers of the TCP/IP stack. In this way, a three-way handshake is performed and the request is dispatched to another server with no problem.

The hash table of transferred connections is implemented through a Linux *slab cache* [14] of pre-allocated structure elements. This design choice allows for very fast allocation and release operations, which are quite frequent under heavy traffic.

We have chosen to implement the forwarding mechanism just under the IP level of the TCP/IP stack. In this way, client packets towards the Web servers must not travel the TCP/IP stack of the switch node, thus avoiding costly checksum (re)computation.

4.3 THOP Communication Protocol

The entire TCP connection transfer mechanism is synchronized through control messages. To this purpose, we have implemented a new communication protocol (THOP) in the standard TCP/IP stack. The THOP protocol defines a small number of messages, that are encapsulated into IP datagrams. Each message triggers an appropriate action or notifies an event to the destination. Let us describe its key role in the TCP handoff mechanism.

To minimize the processing overhead associated to communications, we chose to make the THOP protocol as light as possible. THOP has a (albeit very short) header, made up of two 16 bit fields: the *opcode* identifies the message type, the *checksum* is used to verify the message integrity. The message body depends on the message type. The current implementation supports the following messages.

The *THOP_CREATE* message encapsulates all information necessary for transferring a socket to a Web server. It contains the parameters of the *tcp_opt* structure (which stores the state of a TCP connection) and information about the client, such as the IP source address, the TCP source port and the content of the HTTP request.

The *THOP_NOTIFYCLS* message is sent from a Web server when the corresponding TCP connection is closed. The Web switch does have to know when a TCP connection is closed, because it must delete the corresponding entry from the table of transferred connections. The only way a Web switch knows that a TCP connection is closed without receiving an explicit message is through a TCP reset from a client. The reset is forwarded to the server which takes care of releasing the associated system resources.

The *THOP_NOTIFYDROP* message is sent from the Web server to the Web switch when the duplication of a TCP connection has not been possible for whatever reason (typically, the maximum number of outstanding connection requests has been reached).

We have previously stated that the server does not require any kind of configuration for communicating with a switch node. This is possible because, at connection transfer time, the source IP address in the IP header of the Handoff request is stored into a field of the newly created socket, namely *swaddr*. The *swaddr* field indicates whether a socket has been established through an Handoff or through a three-way handshake. In the former case, it stores the switch IP address; in the latter case, it stores an empty value.

It is worth observing that the THOP protocol provides a very flexible communication mechanism among the nodes of the Web cluster. For example, it is not only suitable to synchronize the con-

nection transferring process, but it may be also extended to implement a distributed load communication mechanism.

4.4 Client Transparency

For the handoff mechanism to be transparent to clients, the source IP address and the TCP timestamp option inserted into server response packets must be modified in such a way that they seem to be generated by the Web switch.

The modification of the source IP address is performed in the following way. Each time a TCP connection is cloned at the server, a special field in the TCP control block, *saddr*, is filled with the source IP address of the Web switch. Instead, ordinary TCP connections leave this field empty. When the server sends data over a TCP connection, it checks the value of the *saddr* field in the corresponding TCP control block. If it is not empty ***** FIXME *** come si dice non nullo?**, the TCP connection has been cloned through a TCP Handoff. In this case, the source IP address is changed with the value contained in the *saddr* field.

Another critical issue in preserving client transparency concerns the use of the TCP PAWS algorithm [17]. The use of a timestamp (encapsulated as a TCP option) has proven very useful for the detection of segments carrying wrapped sequence numbers. As a consequence, the PAWS algorithm is adopted by many TCP implementations, and must be taken into account when sending responses transparently to the clients. However, a key requirement of the PAWS algorithm is that timestamps be always non decreasing. This is not guaranteed in a clustered environment where switch and server nodes have their own, independent timestamps. It must be guaranteed that the timestamp written in the response packets be greater than the value stored at the client. In our Web switch, this is achieved through the following approach. The server uses its own timestamp; in this way, the semantics of PAWS is preserved. Instead, the server transforms its timestamp into a value that is compatible with the value expected by the client from the switch. The transformation is performed as a two-stage process which is illustrated shortly. Let T_{sw}^{tr} be the TCP timestamp value at the switch node, when the TCP connection is transferred. Moreover, let T_{se}^{acc} be the TCP timestamp value at the server when the TCP connection is accepted and T_{se}^{trn} the TCP timestamp value at the server when a packet is transmitted. As soon as the server accepts a connection request, it computes the temporal difference between both timestamps:

$$\delta = max(0, T_{sw}^{tr} - T_{se}^{acc} + 1) \quad (2)$$

A non positive value of δ implies that $T_{sw}^{tr} < T_{se}^{acc}$; hence, the client sees a non decreasing timestamp and there is no need for spoofing it. A positive value of δ indicates that $T_{sw}^{tr} \geq T_{se}^{acc}$. By using the server TCP timestamps would confuse the PAWS algorithm. For this reason, when the server transmits a packet, it writes into the packet the following, augmented timestamp T_{se}^{aug}:

$$T_{se}^{aug} = T_{se} + \delta \quad (3)$$

Finally, when the server receives a client packet containing a timestamp T_{sw}, it must convert that value (which, we recall, is coherent with the switch) into its real value T_{se}:

$$T_{se} = T_{sw} - \delta \quad (4)$$

If $\delta = 0$, there is no need of converting the timestamp at all. Otherwise, if $\delta > 0$, the correct value is deduced.

4.5 Operations of the Web switch

To describe the operations of the proposed one-way Web cluster, we refer to Figure 3, that for the sake of clarity does not show the creation and closure of a TCP connection.

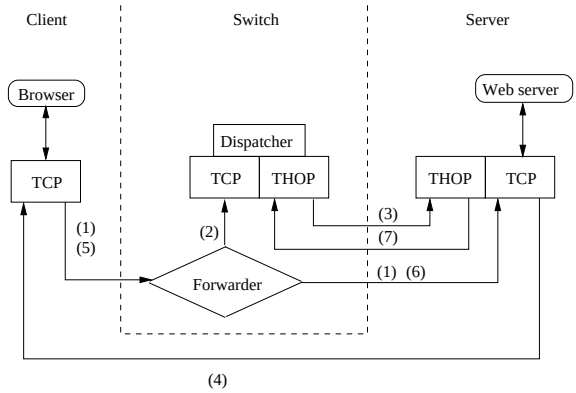

Figure 3: Inner operations of the Web switch.

The first part of the HTTP request sent by the client (1) is intercepted by the forwarder module on the Web switch (2). The forwarder module delivers the request to the higher layers. The TCP protocol issues a call to the dispatcher module upon the arrival of the application-layer data. The dispatcher parses the HTTP request and chooses a Web server according to some content-aware dispatching policy (e.g., LARD, CAP, FLEX). The TCP protocol builds a THOP_CREATE message and sends it to the chosen Web server (3). The identifiers of the TCP connection, such as IP addresses and TCP ports, are packed into a structure and inserted into the hash table containing (active) transferred connections. In such a way, the Web switch is able to forward successive client packets referring to already established connections. The Web server may reply with a THOP_NOTIFYDROP, if it is not able to fulfill the request for whatever reason. In such a case, the Web switch removes the appropriate entry from the mapping table and aborts the TCP connection. In most cases, the Web server accepts the TCP connection, hence it builds the response and starts sending data directly to the client (4). Client ACKs sent to the Web switch are intercepted by the forwarder module of the switch (5), which analyzes the IP and TCP headers to extract all information necessary to perform the lookup in the mapping table. If an entry is found in the table, the packet is forwarded to the previously chosen Web server (6). As the server closes the TCP connection, it notifies this event to the Web switch with a THOP_NOTIFYCLS message (7). The Web switch removes the appropriate entry from the mapping table of transferred connections.

The limit of the present implementation is that only one Web switch may be active at a time in the Web cluster. However, for availability purposes, it is possible to configure two machines in the LAN as Web switches and let one behave as a backup in the case of failures of the first machines.

4.6 Modification to the TCP finite state machine

The TCP Handoff mechanism implemented in our Web cluster requires some modifications to the TCP/IP stack protocol. Figure 4 shows the modified finite state machine. For clarity reasons, this figure does not include the *SYN_SENT* state that is not modified.

Let us first describe the modifications concerning the Web switch. The scheme in Figure 4 applies to sockets which are subject to TCP handoff. The other sockets are handled as usually. With respect to the original TCP finite state machine, we have added two new states: *WAITHEADER* and *FORWARDING*.

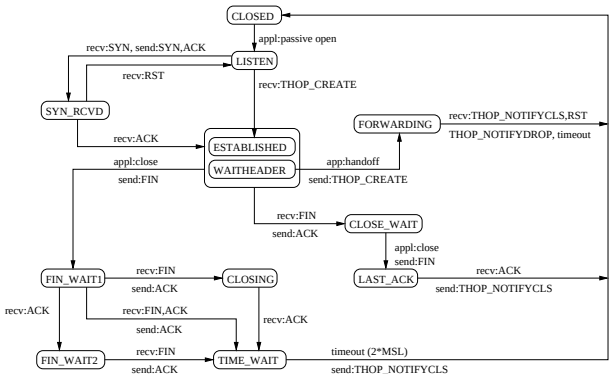

Figure 4: Modified TCP finite state machine.

The WAITHEADER state is equivalent to ESTABLISHED, with the following differences. In the WAITHEADER state, the content of the client request and not only the TCP header field is analyzed. The WAITHEADER state is entered at the end of the three-way handshake, if handoff is enabled for the specified socket; it is left not only when the socket is closed, but also when TCP handoff is carried out. In the last case, the TCP connection is moved to the new state FORWARDING. The FORWARDING state is introduced to inform the Web switch that TCP packets are being forwarded to another server.

On the Web server operating system, the only difference with respect to the original TCP protocol occurs at the establishment of a new connection. Indeed, a new TCP connection may also be created upon the receipt of a THOP_CREATE message from the Web switch. In this case, there is a direct transition from the LISTEN state to the ESTABLISHED state, thus bypassing the three-way handshake procedure which has already carried out between the client and the Web switch. The socket structures on the Web servers contain a further information, namely the IP address of the Web switch. This solution is necessary to implement the communications between the Web switch and the server.

5. EXPERIMENTAL RESULTS

The analysis of the Web cluster performance is divided in two main parts. First, we try to evaluate the overheads of the Handoff mechanism and the scalability of the proposed content-aware Web switch.

Finally, we analyze the performance of the Web cluster for different dispatching policies under a realistic workload.

5.1 Testbed architecture

The Web cluster consists of a Web switch node, and up to seven Web servers. Each machine is a Dual PentiumIII-833Mhz PC with 512MB of memory. All nodes of the cluster use a 3Com 3C905C 100bTX network interface. They are all equipped with a Linux operating system (kernel release 2.4.18). Apache 2.0.43 is used as the Web server software. Dynamic pages are implemented by means of PHP scripts ***** FIXME *** cambiare con CGI executables** which stress various system components, such as the CPU or the disk. The clients and servers of the system are connected through a switched 100Mbps Ethernet.

We used a modified version of the *httperf* tool [20] (version 0.8). This software has been enriched with a support for measuring the 90-percentile and cumulative distributions of the response time.

5.2 Overheads of the Web switch

In an overhead analysis of the Web switch, it is important that the Web servers provide the fastest response possible. For this reason, the clients request one small-sized, static file that is always served by the disk caches. This choice is motivated by the need of finding out how much overhead the switch imposes over the standard TCP protocol. Indeed, the small size of the request adapts quite well to the successive scalability analysis, in which we are interested in seeing how quickly the Web switch is able to transfer connections.

To quantify the overhead of the content-aware mechanisms, we measure the response time of a request provided by a server that is directly connected to the client without Web switch. We then perform the same experiments by including the proposed Web switch between the client and the server. We consider two hardware architectures for the Web switch: the first based on one CPU, the second based on two SMP CPUs.

We are also interested in evaluating the overheads of the Web switch components when the size of the requested file changes. To this purpose, we repeated the above test for different file sizes.

Figures 5 and 6 report the mean response times and the throughputs as a function of the number of HTTP/1.0 (equivalent to TCP) connections. We notice that the proposed content-aware mechanism (ClubWeb-1w) has a negligible impact on the overall performance of a Web server.

In particular, as long as the network does not become the bottleneck of the whole system (that is, when the file size is less than 5Kb), on a dual-CPU Web switch the maximum response time increases linearly up to 6.71% (at 3Kb). It is worthy to note that with a very small file size (that can fit into a single IP packet), ClubWeb-1w does not show any significant overhead with respect to the single server. We ran different tests for file sizes greater than 5Kb, and we verified that in all cases that the network capacity was saturated (89.1 Mbps). Moreover, the switch overhead kept at around 10%-16%.

Figure 6 shows that ClubWeb-1w is able to handle almost the same throughput as a single server. This is another indication that our proposed content-aware Web switch is able to transfer TCP connections (and to forward packets to them) with a very limited overhead.

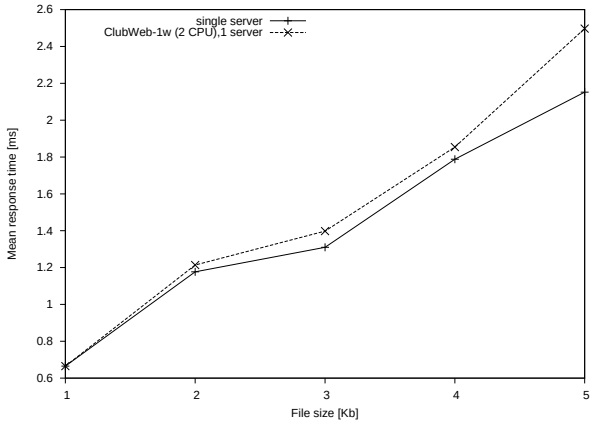

Figure 5: Response time of ClubWeb-1w versus single server.

5.3 Scalability analysis

As a result of the previous stress analysis, we can confirm that the mechanisms behind ClubWeb-1w are very efficient, even when the network is close to saturation. We now evaluate the scalabil-

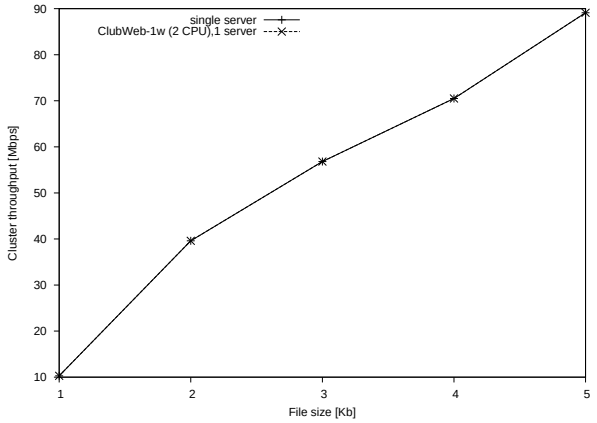

Figure 6: Throughput of ClubWeb-1w versus single server.

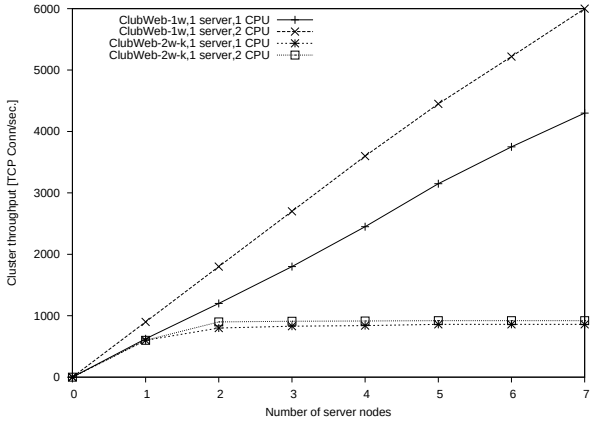

Figure 7: Scalability of content-aware architectures.

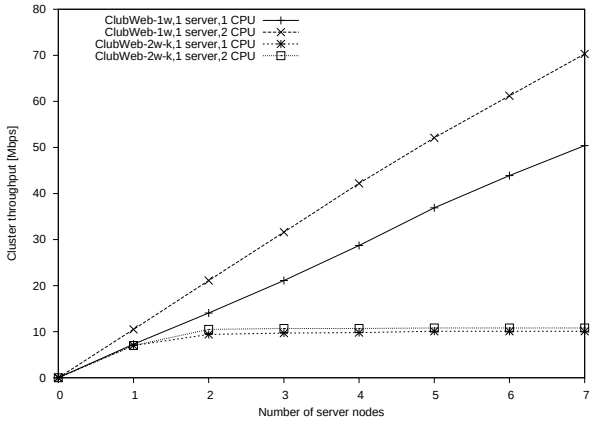

Figure 8: Scalability of content-aware architectures.

ity of the proposed content-aware mechanisms for a higher number of server nodes in the Web cluster. To this purpose, we have implemented a *Round Robin* dispatching policy, which maps different client requests circularly. We request a small-sized, static file (ca. 1500 bytes, the home page of the Apache Web server). We compare ClubWeb-1w with another two-way kernel-level solution that we implemented previously, *ClubWeb-2w-k* [1]. The comparison is performed both on a single-CPU machine and on a dual-CPU machine. The results of these experiments are reported in Figures 7 and 8, respectively. This figure evidences clearly the scalability limits of the two-way solutions. ClubWeb-2w-k does not scale over two nodes for both single- and dual-CPU architectures, serving up to 900 TCP connections per second. The gain due to an additional CPU is up to 12% with two nodes. Thus, we can conclude that the two-way solutions do not scale across multiple nodes. On the other hand, the scalability of ClubWeb-1w is almost linear without significant performance losses. We have verified that, with more than seven Web server nodes, a dual-CPU Web switch tends to saturate the capacity of the network, thus limiting the scalability of the Web cluster.

These experiments and other not reported because of space limits should clarify that the common belief about the poor scalability of content-aware Web switches concerns two-way architectures. On the other hand, a careful kernel-based implementation of a one-way system that can take advantage of a simple SMP architecture does not seem to have any performance problem. Layer-4 solutions remain one-two orders of magnitude faster, but the question is whether it is really necessary to have those levels of performance for a Web cluster. We think that when a Web site has to manage more than ten thousands connections per second it is better (even for availability reasons) to pass to a different architecture, such as two or more Web clusters distributed over different network locations. For lower throughputs, instead, a one-way layer switch is fine with the additional values of content-aware management.

5.4 Performance results for realistic workload

In the last set of experiments we evaluate the performance of the ClubWeb-1w mechanism under realistic workload conditions. In particular, we also consider two content-aware algorithms (CAP and LARD). We run different tests for HTTP/1.0.

A more complete evaluation under realistic workload conditions requires a quite different workload, which takes into account concepts like user sessions, user think-time, embedded objects per Web page, reasonable file sizes and popularity. Our workload consists of a mix of static and dynamic documents. Table 1 summarizes the probability mass function (PMF), the range, and the parameter values of the workload model for static requests and client behavior.

The dynamic portion of the workload is implemented by means of PHP scripts which emulate various services that are classified in four categories: *Type 1* represents the retrieval of data which is cached at the Web server and does not require intensive CPU or disk usage. *Type 2* represents queries to databases,consuming a high amount of disk resources. *Type 3* represents activities that stresses the CPU, such as ciphering. *Type 4* represents a heavy Web transaction that requires a database query and ciphering of the results. It stresses both CPU and disk resources.

A dynamic service yields different results depending on many side conditions, such as, for example, the time at which the request was issued, the load conditions of the chosen server, and the degree of volatility of the retrieved documents. This affects response times in a sensible way. To take this aspect into account, for each request we pre-generate a random number ranging from 1Kb to 100Kb, representing the size of the buffer on which we perform dynamic computations. We chose not to generate bigger buffers since it has been shown that the heavy-tailed characteristics of file sizes is less evident in the context of dynamic services. We consider two workload scenarios. Both of them consist of 80% of dynamic requests and 20% of static requests. Among the dynamic requests, 50% are of *Type 1*, 25% are of *Type 2*, 12.5% are of *Type 3*, 12.5% are of *Type 4*. In the *light scenario* and in the *heavy scenario* client inter-

Table 1: Workload model for static requests and client behavior.

Category	*Distribution*	*PMF*	*Range*	*Parameters*
Requests per session	Inverse Gaussian	$\sqrt{\frac{\lambda}{2\pi x^3}} e^{\frac{-\lambda(x-\mu)^2}{2\mu^2 x}}$	$x > 0$	$\mu = 3.86, \lambda = 9.46$
User think time	Pareto	$\alpha k^\alpha x^{-\alpha-1}$	$x \geq k$	$\alpha = 1.4, k = 1$
Objects per request	Pareto	$\alpha k^\alpha x^{-\alpha-1}$	$x \geq k$	$\alpha = 1.33, k = 2$
HTML object size	Lognormal	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$	$x > 0$	$\mu = 7.630, \sigma = 1.001$
	Pareto	$\alpha k^\alpha x^{-\alpha-1}$	$x \geq k$	$\alpha = 1, k = 10240$
Embedded object size	Lognormal	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$	$x > 0$	$\mu = 8.215, \sigma = 1.46$

arrival times are exponentially distributed with mean equal to 10 seconds and 1 second, respectively.

***** FIXME *** nuova descrizione modello di carico**

The dynamic portion of the workload is implemented by means of CGI executables which stress well-defined system components, such as the CPU or the disk. We have defined the following two workload scenarios.

CPU-bound 40% of static requests, 40% of lightly dynamic requests, 20% of heavy dynamic requests. The dynamic services stress only the CPU. This scenario emulates CPU-bound services (secure browsing).

DISK-bound 40% of static requests, 40% of lightly dynamic requests, 20% of heavy dynamic requests. The dynamic services stress only the disk. This scenario emulates disk-bound services (search engines).

***** FIXME *** fine nuova descrizione modello di carico**

***** FIXME *** rivedere tutti i commenti**

The results of the experiments for this realistic workload model are reported in Figure 9 (response time) and Figure 10 (throughput in Mbps). There are many types of information that canbe get from these figures. An important premise is that in all tests the CPU utilization of the Web switch was never higher than 0.4, hence the limit of the capacity of the Web clusters are due to the servers, especially for dynamic and secure requests.

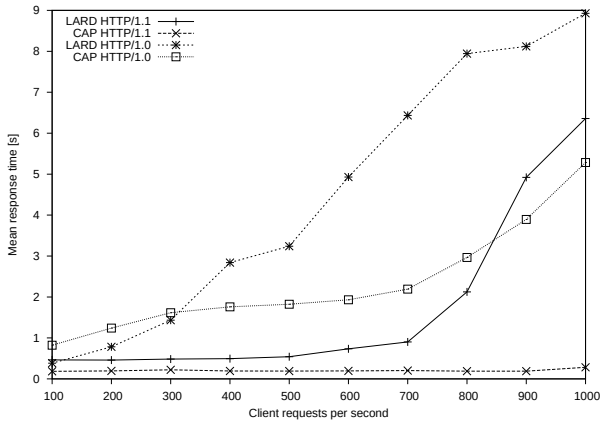

Figure 9: Mean response times (Realistic workload model).

Let first say that both the figures contradict another myth about content-aware Web switches: typical implementations force all the requests from the same client to be directed to the same server, thus preventing scalability because the embedded objects cannot be

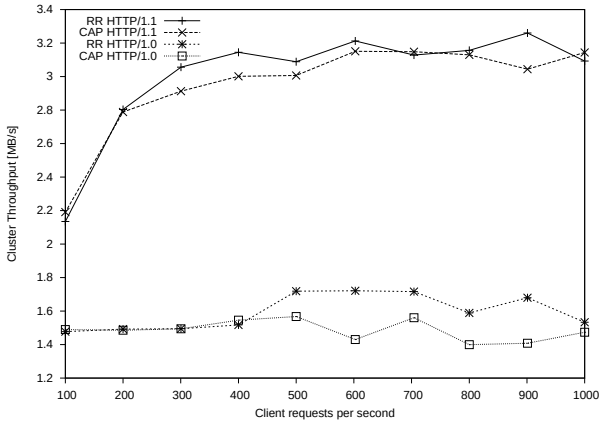

Figure 10: Throughput of the Web cluster (Realistic workload model).

served in parallel, and limiting cache hit rates. All our experiments led us to the following conclusions:

- The HTTP/1.1 protocol is managed in a much faster way by (our implementation of) the Web switch. The persistent connections of HTTP/1.1 allow the Web cluster to achieve a throughput up to 27.2 Mbps, while the HTTP/1.0 protocol does not allow for more than 17.6 Mbps. With HTTP/1.0, each request is sent through a different connection, which must be established and transferred. The associated operations are expensive and limit the performance of the Web cluster. Avoidance of TCP connections handoffs and TCP slow starts provides a substantial performance improvement. With the ever increasing percentage of HTTP/1.1 connections, the performance of Web cluster based on a content-aware Web switch will continue to improve in the future.

- Any attempt to manage the embedded objects of the same persistent client request (in HTTP/1.1) through parallel services leads to poorer performance than that shown here, where all the objects are provided by the same servers. Even the redirection-like mechanism proposed by Aron et al. [3] seems to introduce more overheads than real benefits. We are not aware of other proposals to manage HTTP/1.1 requests in parallel in one-way architectures.

- The only real limit of sending all embedded objects to the same server is that it prevents partitioning of the Web site content among the servers. The other often cited problem of poor cache hit rates is another myth. We have observed that the most popular static objects tend to be present in the disk

caches of all servers. (This is due to the Zipf-like distribution of the popularity of the objects.) Hence, the cache hit rates are not penalized.

- If we pass to dynamic requests, we have to consider that all Web clusters providing complex services that require high performance are organized in multi-tiers. The dynamic requests are managed by back-end servers, not by the Web serves, and there are one or more levels of switching between the Web servers and the back-end serves. Hence, the computationally expensive requests can be served in parallel by the back-end, even if the entire HTTP/1.1 connection is maintained with one Web server.

6. CONCLUSIONS

In this paper, we have shown the design and implementation of an efficient, content-aware distribution mechanism for Web clusters. In particular, we have shown how careful design and optimized implementation choiches may lead to a Web switch that has limited overheads, expecially on SMP nodes. The experimental results have shown that our solution is extremely scalable, thus making a content-aware Web switch a suitable solution to the performance requirements of the majority of cluster-based architectures.

7. REFERENCES

[1] M. Andreolini, M. Colajanni, and M. Nuccio. Kernel-based web switches providing content-aware routing. In *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, Apr. 2003.

[2] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha. Design, implementation and performance of a content-based switch. In *Proceedings of the 19th IEEE International Conference on Computer Communications (INFOCOM 2000)*, pages 1117–1126, Tel-Aviv, Israel, Mar. 2000.

[3] M. Aron, P. Druschel, and Z. Zwaenepoel. Efficient support for P-HTTP in cluster-based Web servers. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 185–198, Monterey, CA, June 1999.

[4] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[5] Array Networks Inc. `http://www.arraynetworks.net`.

[6] L. Aversa and A. Bestavros. Load balancing a cluster of Web servers using Distributed Packet Rewriting. In *Proceedings of the 19th IEEE International Performance, Computing, and Communication Conference*, pages 24–29, Phoenix, AZ, Feb. 2000.

[7] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server system. *ACM Computing Surveys*, 34(2), June 2002.

[8] E. Casalicchio and M. Colajanni. A client-aware dispatching algorithm for Web clusters providing multiple services. In *Proceedings of the 10th International World Wide Web Conference*, pages 535–544, Hong Kong, May 2001.

[9] L. Cherkasova and M. Karlsson. Scalable Web server cluster design with WARD. In *Proceedings of the 3rd International Workshop on Advanced issues of E-Commerce and Web-Based Information Systems*, pages 212–221, San Jose, CA, June 2001.

[10] Cisco Systems Inc. `http://www.cisco.com/`.

[11] M. Dahlin. Interpreting stale load information. *IEEE Trans. Parallel and Distributed Systems*, 11(10):1033–1047, Oct. 2000.

[12] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable and highly available Web server. In *Proceedings of the 41st IEEE Computer Society International Conference*, pages 85–92, San Jose, CA, Feb. 1996.

[13] F5 Networks Inc. `http://www.f5labs.com/`.

[14] B. Fitzgibbons. The linux slab allocator, Oct. 2000. `http://www.cc.gatech.edu/people/home/bradf/cs7001/proj2/`.

[15] Foundry Networks Inc. `http://www.foundrynet.com/products/webswitches/serveriron/`.

[16] IBM. IBM WebSphere Edge Server. `http://www.ibm.com/software/webservers/edgeserver/`.

[17] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*. RFC 1323, May 1992.

[18] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias. Design and performance of a Web server accelerator. In *Proceedings of the 18th IEEE International Conference on Computer Communications (INFOCOM'99)*, pages 135–143, New York, NY, Mar. 1999.

[19] Lucent Technologies. Lucent Web Switch. `http://www.bell-labs.com/project/webswitch/`.

[20] D. Mosberger and T. Jin. httperf - A tool for measuring web server performance. In *Proceedings of Workshop on Internet Server Performance*, Madison, Wisconsin, 1998.

[21] Nortel Networks Ltd. Nortel Networks Web OS. `http://www.nortelnetworks.com/products/01/alteon/`.

[22] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, CA, Oct. 1998.

[23] J. Postel. *Transmission Control Protocol*. RFC 793, Sept. 1981.

[24] Radware Inc. `http://www.radware.com/`.

[25] A. Rijsinghani. *Computation of the Internet checksum via incremental update*. RFC 1624, May 1994.

[26] P. Russell. Unreliable guide to locking, 2000. `http://netfilter.gnumonks.org/unreliable-guides/kernel-locking.a4.ps`.

[27] J. Song, A. Iyengar, E. Levy-Abegnoli, and D. Dias. Architecture of a Web server accelerator. *Computer Networks*, 38(1):75–97, Jan. 2002.

[28] C.-S. Yang and M.-Y. Luo. A content placement and management system for distributed Web-server systems. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 691–698, Taipei, Taiwan, Apr. 2000.

[29] Zeus Technologies Ltd. `http://www.zeus.com/`.

[30] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. HACC: An architecture for cluster-based Web servers. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 155–164, Seattle, WA, July 1999.