Kernel-based Web switches providing content-aware routing

Mauro Andreolini
Department of Information, Systems and Production
University of Roma 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
nuccio.marcello@unimo.it

Abstract

Locally distributed Web server systems represent a cost-effective solution to the performance problems due to high traffic volumes reaching popular Web sites. In this paper, we focus on architectures based on layer-7 Web switches because they allow a much richer set of possibilities for the Web site architecture, at the price of a scalability much lower than that provided by a layer-4 switch. In this paper, we compare the performance of three solutions for layer-7 Web switch: a two-way application-layer architecture, a two-way kernel-based architecture, and a one-way kernel-based architecture. We show quantitatively how much better the one-way architecture performs with respect to a two-way scheme, even if implemented at the kernel level. We conclude that an accurate implementation of a layer-7 Web switch may become a viable solution to the performance requirements of the majority of cluster-based information systems.

1 Introduction

The ever increasing demand for complex, efficient services offered through Web interfaces, and the constantly growing number of Web users are putting a serious strain on today's Internet services. Unfortunately, upgrading a single server does not represent a valid solution to the scalability problem, because Web traffic is characterized by a continuous growth. Many solutions based on caching and server replication have appeared to face the scalability problem of Web content delivery. In this paper, we focus on *locally distributed Web-server systems*, briefly *Web clusters*. For a complete survey on the topic, see [7].

A Web cluster is a set of server machines that are hosted together at the same location, and interconnected through a high-speed LAN (see Figure 1).

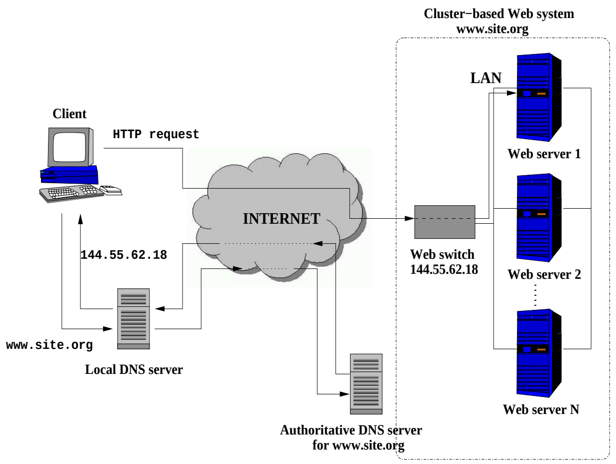

Figure 1. Web cluster architecture.

The cluster is publicized through one site name and one *virtual IP address* (VIP). This is typically the address of a dedicated front-end node (also called *Web switch*) which acts as an interface between the nodes of the cluster and the rest of the Internet, thus making the distributed nature of the site completely transparent to the users and client applications. In this architecture, the authoritative DNS server of the Web site always maps the site name into the VIP address. Hence, the Web switch receives all client requests and routes them to a Web server node through some centralized *dispatching policy*. Web clusters may be classified according to the OSI protocol stack layer at which the Web switch routes inbound requests. We distinguish *layer-4* and *layer-7* switches. A layer-4 Web switch works at TCP layer and performs content-blind routing that is, it does not take into account any content information in the client request when taking dispatching decisions.

On the other hand, layer-7 Web switches can work with application level information, thus allowing for a much richer set of solutions for the Web cluster. For example, in a cluster based on a layer-4 switch, the entire document tree must be replicated among all servers or at least shared through an NFS file server. A layer-7 switch permits content-based routing with consequent *content partitioning* solutions and the possible use of specialized servers [17]. Dynamic content partitioning also improves Web cluster performance by increasing the cache hit rates of single servers [14, 15]. Commercial products typically parse the HTTP request either to partition the servers according to the service type they handle or to provide persistent session support, based on cookies or SSL identifiers. Another important advantage of a layer-7 Web switch is that it can implement more sophisticated admission control algorithms that are of key importance for the next generation of Web clusters, that will pass from high performance to differentiated and guaranteed performance [5, 9, 18].

The drawback is that content-based routing introduces non negligible overheads on the Web switch. This component, if not well implemented, may become the bottleneck of the whole Web cluster. These considerations have seriously limited the diffusion of layer-7 Web switches that cannot guarantee a performance level comparable to that of layer-4 switches. To address this problem, some architectural alternatives are proposed in [15, 6, 17] which try to combine the advantages of content-blind with those of content-aware distribution. This paper aims also to demonstrate that, although layer-7 switch performance cannot be comparable to that of layer-4 switches, an accurate kernel-based Web switch can provide a throughput that is sufficient to the majority of content providers, to the extent that the bottleneck can be moved from the Web switch to the Internet connection of the cluster.

There are two main characteristics to classify layer-7 Web switches. The first concerns the implementation layer: kernel-based or application-based.

The second considers the flow of the packet traffic between the client and the Web cluster. The main difference lies in the backward flow, because all inbound packets must pass through the Web switch. In *two-way* architectures, the outbound packets pertaining to a response pass again through the Web switch. This mechanism can be implemented at the kernel- or application-layer.

In *one-way* architectures, the servers send response packets directly to the client. One-way architectures are expected to be more efficient than two-way solutions, since they limit the risks of system bottleneck at the Web switch due to forward and backward handling of each packet. On the other hand, one-way architectures working with a layer-7 Web switch require a much more complex kernel-based implementation that affects both the switch and each Web server. The motivation is that the distributed architecture of the cluster must remain transparent to both the user and 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 must remain the only official interface for the clients.

Table 1 shows some combinations of alternative architectures for content-aware Web switches, that are proposed in literature and by the same authors: layer-4 vs. layer-7, kernel- vs. application-layer implementation, one-way vs. two-way schemes. The main focus of this paper is to provide a fair comparison among different 7-layer prototypes running on the same platform that is, identical hardware and operating system. The main results are as following.

- We quantify the performance differences between a kernel-based and an application-based implementation of a two-way architecture. We see that a kernel-based Web switch can sustain a number of connections which is significantly higher (almost four times) than that of an application-based switch.

- We then pass to consider the impact of one-way vs. two-way solutions, both of them implemented at a kernel level. We see that one-way architecture performs much better (almost two times) than two-way solutions.

- If we compare a one-way kernel-based solution and a two-way application-based implementation , the performance ratio is as good as eight to one. With similar results, that are obtained through inexpensive PC/Linux machines instead of dedicated network components , layer-7 routing becomes a viable solution for the performance requirements of the majority of Web content providers using cluster-based architectures.

The rest of this paper is organized as following. Section 2 describes the design and implementation of a two-way Web cluster solution operating at the kernel level, while Section 3 outlines the kernel-based one-way architecture. Section 4 evaluates the performance of different Web cluster implementations. Section 5 concludes the paper with some final remarks.

Table 1. Content aware Web switches.

Name	*Scheme*	*Switch implementation*	*Server implementation*	*References*	*Name*
TCP Gateway	two-way	application level	n.a.	[6, 11]	ClubWeb-2w-a
	two-way	kernel level	n.a.	here	ClubWeb-2w-k
TCP Splicing	two-way	kernel level	application level	[6, 10]	
TCP Handoff	one-way	FreeBSD kernel	FreeBSD kernel	[6, 16]	ScalaServer
	one-way	Linux kernel	Linux kernel	[3]	ClubWeb-1w

2 Two-way kernel-based Web cluster

In this section we propose the architectural design of a two-way kernel-based Web switch based on the TCP Gateway scheme, where the switch acts basically as a proxy server. As soon as a client request arrives, the Web switch chooses the Web server by applying some content-aware dispatching policy and transmits the request to it. If necessary, a new TCP connection is established with the server. The response is sent to the Web switch, which directs it back to the client.

The proposed prototype is called two-way cluster-based Web System or *ClubWeb-2w-k*. This architecture, based on the *kHTTPd* server [12], has been designed and implemented as a loadable kernel module in the Linux operating system, kernel version 2.4. For symmetry reasons, the application-based switch based on the TCP Gateway scheme [11] is called *ClubWeb-2w-a*.

The overall design of the Web switch is thread-oriented. A bunch of *main daemons* are responsible for handling client requests. These daemons are spawned by a *management daemon* which acts as the interface between the administrator and the main daemons.

2.1 The management daemon

Figure 2 shows the design of the management daemon, which is responsible for spawning main daemons that handle client requests. The management daemon is created as soon as the kernel loadable module is inserted by means of the *insmod* command. It then waits for a specific *start* command which enables Web switch operations. Once the cluster is activated, the necessary amount of memory for all future main daemons is allocated. The allocation is left to the management daemon and not to each main daemon because this makes the overall design simpler and avoids potential troubles due to resource synchronization among the main daemons. Once the memory is allocated, a server socket is initialized to accept client requests. Then, the preconfigured number of main daemons is created and initialized. The management daemon waits for further configuration commands. During this period, some threads which exited abnormally (e.g., due to a SIGKILL) are generated

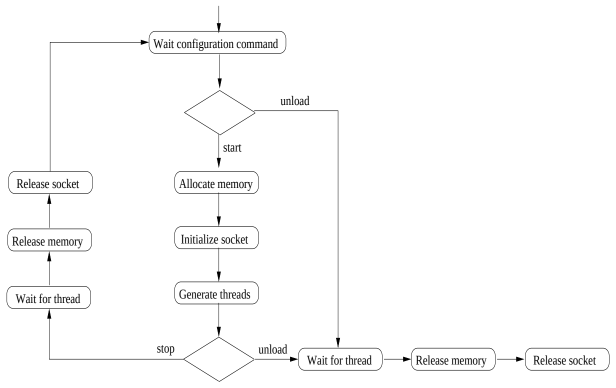

Figure 2. The Management daemon.

again. This ensures that a sufficient number of main daemons is always active. Another duty of the management daemon is that of permitting the selection of the Web switch dispatching policy, which may be chosen even at runtime.

A *stop* command causes the clean termination of all main threads. The server socket is then released. An *unload* command de-allocates the main memory and unloads the kernel module.

2.2 The main daemon

The main daemon is responsible for handling client HTTP requests. Generally, more main daemons are spawned by the management daemon in order to enhance the performance of the Web cluster. A main daemon is created as soon as a *start* command is issued. After the creation, the main daemon enters the wait queue which is associated with the listening socket, and it is woken up at the occurrence of one of the following events: suspension timeout, data arrival, signal delivery.

A main daemon is periodically woken up (after a suspension timeout) for checking the presence of any data or some administration command. Once reached the socket wait queue, the main daemon enters the request handling loop, which is left only when the Web cluster is shutdown or a serious error occurs.

The request handling loop follows an event-driven design. The operations executed for handling client requests are subdivided into *phases*. A main daemon is not oriented to satisfy one client request at a time, but all requests that exist in a given phase. To this purpose, each main daemon has a wait queue for each phase. All requests queued at the first phase are processed before handling requests at the second phase. Clearly, requests having their first phase completed are moved into the second phase queue. This pipeline-oriented approach is much more efficient than that presented in Apache [4], where each thread (or process) would handle sequentially the phases of a single client request. When a client request leaves the last phase of the pipeline, it can be considered completed; its per-request allocated resources are released.

Each main daemon can access only its own set of queues. This implies that a thread is responsible for the execution of all phases of a client request. Sharing queues among multiple threads would introduce synchronization and concurrent access issues that severely affect the Web switch performance. A brief description of the different phases follows.

In the *AcceptConnections* phase, the main daemon checks the presence of connection requests on the socket, and accepts them. For each inbound client request, the necessary data structures are allocated and initialized. Then, the accepted requests are queued into the next phase queue. If the number of active TCP connections exceeds the preconfigured threshold, further client connection requests are refused. This is necessary to prevent overload situations and DoS attacks.

In the *HandleRequest* phase, the main daemon waits for a client HTTP request. If no data is available, it processes every queued request. Once a client request is available, it is parsed to extract application layer information which is necessary for content-aware routing. The request is then queued into the next phase. Malformed or too large HTTP requests are immediately dropped.

In the *HandlePolicy* phase, a Web server is chosen according to the previously parsed application content. If the Web servers are overloaded, the request is dropped.

In the *HandleDispatch* phase, a TCP connection is opened with the chosen Web server for sending the client request. From now on, the Web switch acts as a gateway between the client and the Web server, in the sense that it handles all traffic between the two end-points. When one of the two peers closes the connection, the Web switch handles the connection by closing it at the other peer, and frees up the allocated system resources.

An event-driven approach has several advantages, but has the following main problem. If a main daemon is busy at a given phase, it is not able to handle requests waiting in other phases, thus limiting responsiveness. For this reason, we have introduced a bound on the maximum number of accepted connections in the AcceptConnections phase.

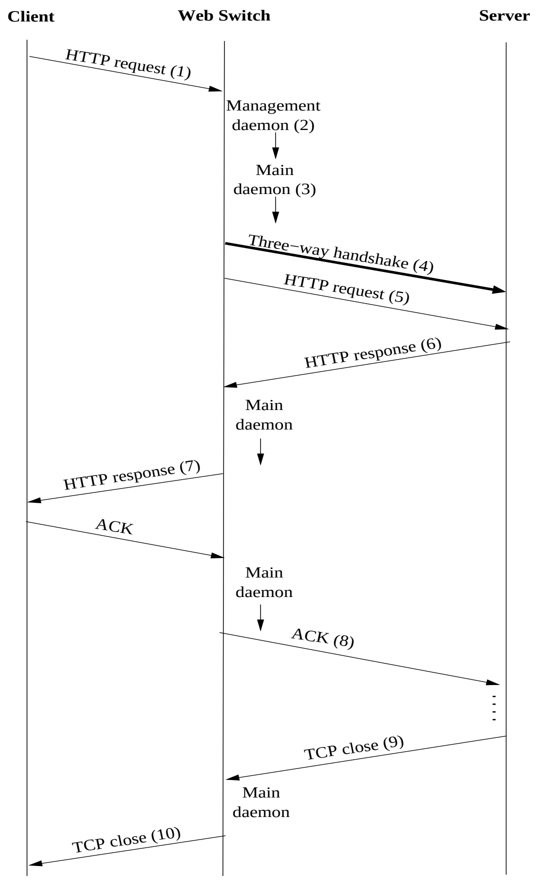

Figure 3. Operations of the TCP Gateway mechanism.

When this threshold has been reached, the corresponding main daemon handles the next phase and does not accept connections anymore (although it would be possible). This mechanism increases the responsiveness of the entire Web cluster because the flow among pipeline phases is faster. Moreover, concurrent main daemons accept connections simultaneously; if one main daemon is blocked at one phase, it would unbalance the load of the other main daemons.

2.3 The TCP Gateway mechanism

We present a detailed description of the operations performed by the TCP Gateway mechanism in ClubWeb-2w referring to Figure 3.

As soon as an HTTP request sent by the client (1) is received at the Web switch, the management daemon assigns a new main daemon to it (2). The main daemon selects a Web server according to some content-aware dispatching policy (3) and establishes a TCP connection with that server (4). Then, the HTTP request is sent to the chosen Web server over the newly established connection (5). In case of a server failure, the main daemon closes the client-switch connection. However, in most cases, the Web server builds the response and starts sending data to the Web switch (6). The main daemon directs those response packets to the client (7). Client ACKs to response packets are sent to the Web switch, which directs them to the appropriate server through the main daemon (8). When the server closes its TCP connection with the Web switch (9), the main daemon closes the corresponding TCP connection with the client (10).

3 One-way kernel-based Web cluster

In this section we outline the functional mechanism of a one-way layer-7 Web switch based on the TCP Handoff approach, also called *ClubWeb-1w*. A detailed technical description of this kernel-based switch can be found in [3]. This prototype is the first that has been designed and implemented as an extension of the TCP/IP stack for the Linux operating system, kernel version 2.4, while a previous implementation was based on the FreeBSD kernel [6].

With one-way architectures, the Web switch accepts a client connection request, parses the HTTP request, chooses a Web server according to a well defined content aware policy and, finally, transfers the TCP connection to that server (along with the application-layer content). Response packets are directly sent to the client by the server that bypasses the Web switch. Client ACKs are sent to the Web switch, which directs them to the appropriate server, in order to preserve the semantics of the TCP protocol.

The main components of the Web cluster (*dispatcher* module, *forwarding* module and the *THOP* handoff protocol module) are summarized below.

The dispatcher module is responsible for parsing the HTTP requests and choosing a Web server according to some content-aware dispatching policy. The dispatcher is invoked during the processing of TCP segments containing application-layer requests on a configurable port.

The forwarder module intercepts Ethernet frames belonging to an already transferred TCP connection and transmits them to the appropriate Web server. This operation requires the extraction of IP addresses and TCP port fields from each incoming frame. This information is used to access a hash table of already transferred TCP connections. If a connection is found in that table, the frame is sent directly to the destination server. Otherwise, it is sent to the higher levels of the TCP/IP stack for usual processing, because it is either a new connection request or a frame for another service. The solution adopted by the forwarder module reduces the overhead due to TCP/IP processing (mainly, checksum verifications).

The THOP protocol implements the necessary messages for synchronizing the phases of a TCP connection transfer between the Web switch and a server. A THOP packet is encapsulated into an IP datagram. It contains a given message which triggers an appropriate action or notifies an event at the receiver. The current implementation defines messages for the following operations:

- Encapsulation and transmission of the complete state of a TCP connection (including the client request) to a given Web server.
- Notification of *TCP connection close* from a Web server, so that the Web switch can delete the corresponding entry from the table of transferred connections.
- Notification of *TCP connection duplication failure* from a Web server. This message may be used also as a notification of server overload.

A description of the operations performed by the TCP Handoff mechanism in ClubWeb-1w follows. The first part of the HTTP request sent by the client is intercepted by the forwarder module on the Web switch, as the TCP connection has not yet been transferred. 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, such as LARD [14], CAP [8], FLEX [1]. The TCP protocol sends a message to the chosen Web server, containing the TCP connection state. The identifiers of the TCP connection, such as IP addresses and TCP ports, are packed into a structure and inserted into a mapping 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 drop message, if it is not able to fulfill the request for whatever reason. In this case, the Web switch removes the appropriate entry from the mapping table and aborts the TCP connection. In the majority of cases, the Web server accepts the TCP connection, then it builds the response and starts sending data directly to the client. Client ACKs sent to the Web switch are intercepted by the forwarder module of the Web switch, 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, the packet is forwarded to the corresponding Web server. As the server closes the TCP connection, it notifies this event to the Web switch with an appropriate THOP message. The Web switch removes the appropriate entry from the mapping table of transferred connections.

It is worth noting that, unlike the ScalaServer architecture [6], no control connection between the Web switch and the server is used. This allows us for a lighter TCP connection transfer that consumes less resources. The limit of our solution 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 switches and let one behave as a backup machine, if the first one fails.

4 Experimental results

In this section we evaluate the performance and scalability of the two proposed kernel-based Web cluster architectures and compare them with the basic TCP Gateway solution. The analysis of the Web cluster performance is divided in two main parts: first, we compare the overheads due to application-based vs. kernel-based solutions and the overheads due to one-way vs. two-way mechanism. Then, we analyze the performance of the Web cluster under a realistic workload.

Every prototype is implemented on inexpensive PC/Linux machines. Hence, performance comparisons with dedicated (and expensive) commercial products are beyond the scope of our research. Other possibly interesting comparisons with public domain solutions, especially the ScalaServer architecture [6], were impossible because of incompatibility of their kernels with the versions for present PC hardware.

4.1 Testbed architecture

We have implemented both ClubWeb-2w-k and ClubWeb-1w as extensions of the Linux kernel, release 2.4.17, while ClubWeb-2w-a has been implemented as a Reverse Proxy Server through the Apache module *mod_rewrite* [11]. The testbed architecture is based on off-the-shelf hardware and software components. The clients and servers of the system are connected through a switched 100Mbps Ethernet. The cluster is made up of a Web switch and up to five 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, while Apache 1.3.26 is used as the Web server software.

We used a modified version of the *httperf* tool [13] (version 0.8). This software has been enriched with a support for measuring the 90-percentile and cumulative distributions of page response times.

4.2 Overhead analysis

For the overhead and scalability analysis of the Web switch, it is important that the Web servers provide the fastest response possible. For this reason, the clients request one static file of approximately 1500 bytes using an increasing number of HTTP/1.0 (equivalent to TCP) connections. In this way the file 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. To quantify the overhead of the content-aware mechanisms, we also show the response time of a request provided by a server that is directly connected to the client without Web switch.

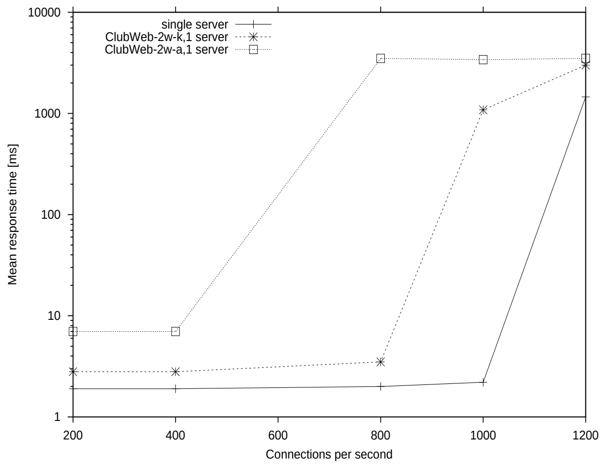

Figure 4. Overheads of kernel-based vs. application-based mechanism.

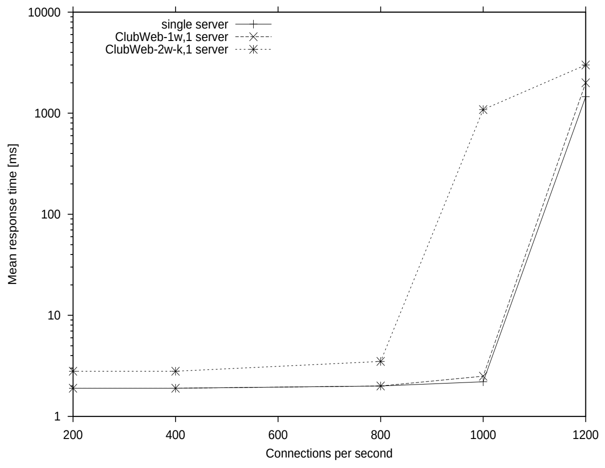

Figure 5. Overheads of one-way vs. two-way mechanism.

The results shown in Figure 4 confirm the great benefits of a kernel-based implementation over an application-based one. We have chosen to compare only two-way solutions, so the difference is imputable mainly to the overhead of context switching in the application-based solution. The switch of ClubWeb-2w-a thrashes at 400 connections per second, while ClubWeb-2w-k limits its performance at 800 connections per second (that is, the double with respect to the application-based solution). This is indicative of the performance difference obtainable from a pure kernel implementation. Moreover, considering the performance of both switches in the non-saturation zone (that is, at a connection rate lower than 400 connections per second), the performance ratio between kernel-based two-way implementation and application-based two-way implementation is approximately four to one. Finally, the overhead of a content-aware two-way solution with respect to a single server is at least of a factor two. We confirm the intuition that two-way solutions introduce a sensible overhead at the Web switch, due mainly to the processing of response packets.

Figure 5 reports a comparison between one-way and two-way mechanisms. We notice that the proposed content-aware mechanism (ClubWeb-1w) has a negligible impact on the overall performance of a Web server. The overhead of ClubWeb-1w is quite low with respect to the single server: it reaches a maximum of 11% at the knee of the curve (1000 connections per second). This result is remarkable because it shows the efficiency of our Web switch implementation even on SMP architectures, that to the best of our knowledge has never been reported in literature. It is interesting to note that, even when the server is under critical load conditions, the mechanisms of ClubWeb-1w allow the system to follow the performance behavior of the single Web server architecture with no intermediaries. ClubWeb-2w-k has lower capacity (it thrashes at 800 connections per second, 20% less than ClubWeb-1w) and shows a significant overhead (72%) in its non-saturation zone (that is, at less than 900 connections per second). We thus confirm (in the best case, with a very small file) the overheads of a two-way mechanisms with respect to a one-way solution. The situation worsens when the requested file sizes are bigger, because of the greater size of the server responses.

4.3 Performance results for realistic workload

In the following set of experiments we evaluate the performance of the proposed architectures under realistic workload conditions, including user sessions, user think-time, embedded objects per Web page, reasonable file sizes and popularity. We run different tests using the HTTP/1.0 protocol, and the Client Aware Policy (CAP) as the dispatching algorithm [8]. The workload consists of a mix of static and dynamic documents. The dynamic portion of the workload is implemented by means of CGI scripts which emulate various services: queries to databases, cyphering, and e-commerce transactions. A more detailed description

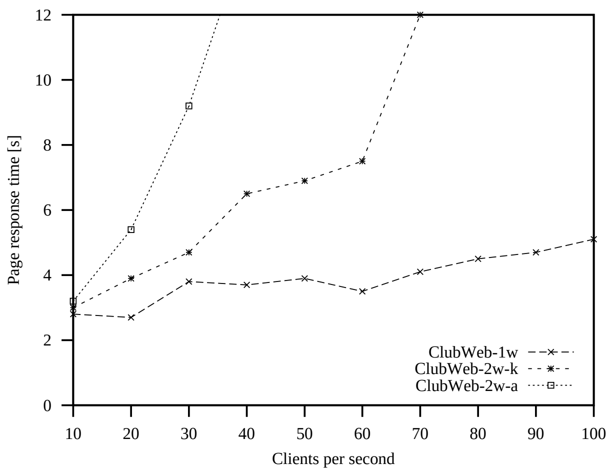

Figure 6. 90-perc. of page response time.

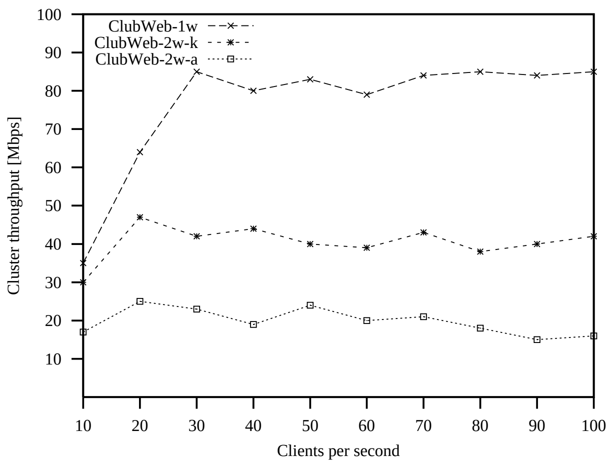

Figure 7. Throughput of the Web cluster.

is given in [2].

The experimental results for increasing number of client requests are reported in Figure 6 (90-percentile of page response time, including the base HTML file and its embedded objects) and Figure 7 (throughput in Mbps). An important premise is that in all tests the CPU utilization of the ClubWeb-1w Web switch was never higher than 0.4, hence this switch never represents the bottleneck of the whole system.

From the figures we deduce that ClubWeb-1w is able to reach a throughput of 85 Mbps, that actually corresponds to the maximum real throughput of the testbeb network (100Mbps Ethernet). Hence, the bottleneck of the cluster is represented by the network capacity. This behavior is even more evident for the HTTP/1.1 protocol, that we cannot report due to space limitations.

The performance gap between kernel-based and application-based solutions is also evident. ClubWeb-2w-a is affected by context-switching overheads which make the switch the cluster bottleneck. This, in its turn, impacts on the cluster throughput, which is degraded of at least a factor of two with respect to the kernel-based solutions.

The performance gap between one-way and two-way architectures is even more pronounced. The saturation of the two-way prototypes limits their throughput to at least one fourth with respect to the one-way solution.

5 Conclusions

The importance of content-aware routing lies in the possibility of implementing sophisticated request dispatching algorithms and access control policies, that are important for present and future cluster-based Web architectures. On the other hand, content-aware mechanisms implemented at the front-end Web switch have been always considered a limit on the scalability of the Web clusters. We have presented the high level design of a one-way and a two-way Web cluster, that are implemented at the kernel level. Our experimental results confirm that two-way architectures where both inbound and outbound packets flow through the Web switch have a system bottleneck in this front-end. On the other hand, we show that with current off-the-shelf PC hardware it is possible to implement a one-way Web cluster that performs very well under realistic workload conditions. This moves the bottleneck of the system throughput from the Web switch to the Internet connection, at least in the large majority of Web clusters that for economic reasons does not use bandwidth connections larger than T3 or OC3.

References

[1] V. Agarwal, G. Chafie, N. Karnik, A. Kumar, A. Kundu, J. Shahabuddin, and P. Varma. An architecture for virtual server farms. Research Report RI 01006, IBM Research, Apr. 2001.

[2] M. Andreolini, M. Colajanni, and R. Morselli. Performance study of dispatching algorithms in multi-tier web architectures. *ACM Sigmetrics Performance Evaluation Review*, 30(2):10–20, 2002.

[3] M. Andreolini, M. Colajanni, and M. Nuccio. Scalability of content-aware server switches for cluster-based web information systems. In *Proceedings of the 12th International World Wide Web Conference*, Budapest, May 2003.

[4] Apache Server Foundation. Apache HTTP Server Project. `http://www.apache.org`.

[5] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, pages 90–101, Santa Clara, CA, June 2000.

[6] 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.

[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] X. Chen and P. Mohapatra. Providing differentiated service from an Internet server. In *Proceedings of the 8th IEEE International Conference on Computer Communications and Networks*, pages 214–217, Boston, MA, Oct. 1999.

[10] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

[11] R. Engelschall. Load balancing your web site. *Web Techniques Magazine*, 3, May 1998.

[12] kHTTPd: a kernel http daemon. `http://www.fenrus.demon.nl/`.

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

[14] 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.

[15] J. Song, E. Levy-Abegnoli, A. Iyengar, and D. Dias. Design alternatives for scalable Web server accelerators. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 184–192, Austin, TX, Apr. 2000.

[16] W. Tang, L. Cherkasova, L. Russell, and M. W. Mutka. Modular TCP handoff design in STREAMS-based TCP/IP implementation. In *Proceedings of the 1st International Conference on Networking, Lecture Notes in Computer Science 2049*, pages 71–80, Colmar, France, July 2001.

[17] C. S. Yang and M. Y. Luo. Efficient support for content-based routing in web server clusters. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

[18] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation in cluster-based network servers. In *Proceedings of the 20th IEEE International Conference on Computer Communications (INFOCOM 2001)*, pages 679 –688, Anchorage, AK, Apr. 2001.