Università degli Studi di Roma “Tor Vergata”
Dipartimento di Informatica, Sistemi e Produzione

XIV Ciclo del Corso di Dottorato di Ricerca in
“Informatica ed Ingegneria dell’Automazione”

Cluster-based Web Systems: Paradigms and Dispatching Algorithms

Emiliano Casalicchio

Docenti guida:
Prof. Michele Colajanni

Prof. Salvatore Tucci

Coordinatore:
Prof. Daniel Pierre Bovet

Roma, Marzo 2002

ds

Contents

Chapter 1

Introduction

Demands placed on Web-based services continue to grow and Web server systems are becoming more stressed than ever.

Recent history of World Wide Web was characterized from events that polarize the attentions of millions of Web users in the world such as the NASA Pathfinder mission, the Olympic games, the Wimbledon Tennis Championship. All official Web sites related to those events experimented millions of access per day. For example the NASA pathfinder Web site in July 1997 received a peak access per day of 40 million of requests; the Wimbledon Web site received a peak access per day of 125 millions and 282 millions of requests in July 1999 and July 2000 respectively; the Olympic Summer games received a peak of 875 millions of requests per day in September 2000. However, not only special event Web sites experiences this huge amount of requests. The Google Web search engine [69] is an example of daily Web tools that satisfy 100 million of search queries per day (January 2001, Data released by Google Inc.). The Google Inc. provides his Web services through a distributed systems of about 10000 servers divided in 4 data centers (two located in the East coast and two in the West coast).

All the cases mentioned above are very interesting case studies and testbeds for Web system architects and researchers. Special solutions based on geographically and locally distributed Web systems are used to support these very popular Web sites.

Because of the complexity of the Web infrastructure, performance problems may arise in many points during a Web transaction. For instance, they may occur at the network level because of congested Internet routers, as well as at the server level either because of under-provisioned capacity or unexpected surge of requests. Although both network and server capacity have improved in recent years, and new architectural solutions have been deployed, response time continues to challenge Web system related research.

In this thesis, we focus on load management solutions in architectures that aim to reduce the delays due to the Web server site assuming that network issues are examined elsewhere. The main guideline of this thesis is that, the proposed methodology and algorithms have immediate applicability in the Web.

Indeed in a completely distributed system such as Internet, the Web site is the only component that can be under the direct control of the content provider administration. Web clients, Internet backbones and routers, DNS system, and proxy servers are not controllable by a single organization, and any intervention on these components is hard to be applicable because it requires an agreement among several organizations. Another motivation for focusing on the Web system architecture is due to the growing complexity of Web applications and services. Web performance perceived by the end users is increasingly dominated by server delays, especially when contacting busy servers [22]. Recent measures suggest that the Web servers contribute for about 40% of the delay in a Web transaction [72] and it is likely that this percentage will increase in the near future. As a consequence of the changes transforming the Web from a simple communication and browsing infrastructure to a complex medium for conducting personal businesses and e-commerce, the computational load placed by network applications on Web servers continue to grow. A prediction made in 1995 regarding network bandwidth estimated that it would triple every year for the next 25 years. So far, this prediction seems to be approximately correct [70], while the Moore law estimates "just" a double system capacity every 18 months. Other network improvements, such as private peering agreements between backbone providers, the deployment of Gigabit wide-area networks, the rapid adoption of ISDN networks, xDSL lines, and cable modems contribute to reduce network latency. With the network bandwidth increasing about twice as fast as than the server capacity, and augmented complexity of Web-based applications, the future bottleneck is likely to be more on the server side.

1.1 Load management in locally distributed Web systems

A Web cluster needs to appear as a single host to the outside world, so that users need not be concerned about the names or locations of the replicated servers and they can interact with the Web-server system as if it were a single high performance server.

Hence, the basic model must adhere to the architecture transparency requirement by providing a single virtual interface to the outside world at least at the site name level. (Throughout this thesis, we will use `www.site.org` as the example name for the Web site.) This choice excludes from our analysis some architectures such as the *mirrored-server* system, a simple solution that lets users manually select alternative names for a Web site, thereby violating the transparency requirement. The obvious advantage is the simplicity of implementation. However, this architecture has several disadvantages, as it is not user-transparent, end does not allow to control the request distribution, thus leading to poor load sharing because users are not aware about load of the server nodes and network conditions. Unlike the users, we assume that the client applications might be aware of the effects of some dispatching mechanisms, even if we inspect only solutions where the clients do not need any modification to interact with the Web cluster. Hereafter, we refer either to *client* (or *Web browser*) as to the software entity that acts as a user agent and is responsible for implementing all the interactions with the Web server that is, generating the requests, transmitting them to the server, receiving the results from the server and presenting them to the user.

From these premises, the basic Web cluster architecture consists of multiple server nodes, housed together in a single location (and connected through an high speed LAN or a dedicated interconnection network) with one or more mechanisms to spread client requests among the nodes. Each Web server can access all site information, independently of the degree of content replication, however the site may require dedicated nodes for special Web services such as video streaming and data transcoding gateway. The Web system includes also one authoritative Domain Name System (DNS) server for translating the Web site name into one or more IP address(es) and, if necessary, one or more internal routing devices. A high-level view of the basic architecture is shown in Figure 1.1. A router and other network components belonging to the Web system may exist in the way between the system and the Internet. Moreover, it has to be noted that an architecture for modern Web sites consists also of back-end nodes that typically act as data and application servers for dynamically generated information. The main focus of this thesis is on the Web server layer while the techniques concerning content distribution at the back-end layer are outlined in Chapter 10.

We analyze now the main phases to serve a user request to a Web site. Although a user issues one request at a time for a *Web page*, he causes multiple client-server interactions because typically a *Web page* is a multi-part document consisting of a

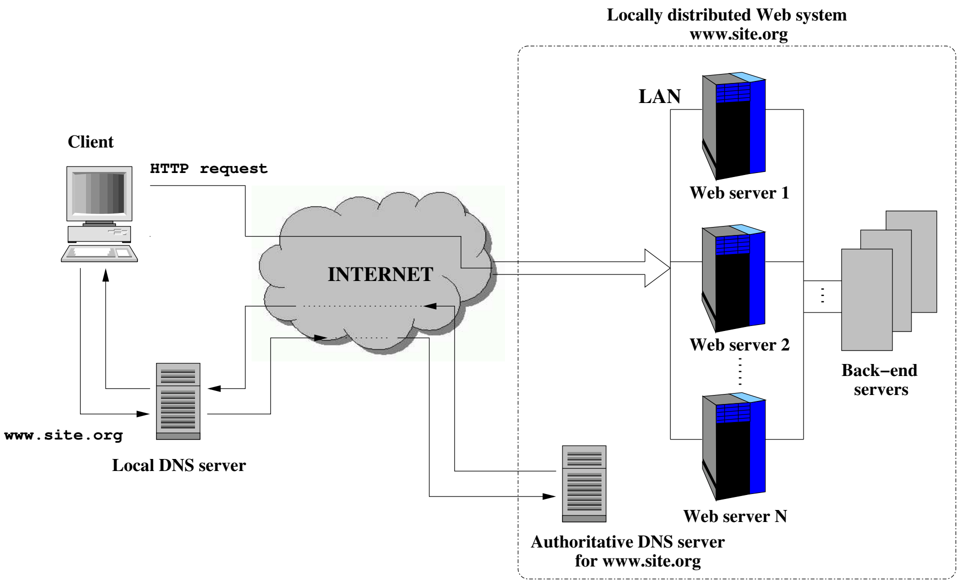


Figure 1.1: Model architecture for a locally distributed Web system.

collection of objects. A static *Web object* is a file in a specific format (e.g., an HTML file, a JPEG image, a Java applet, an audio clip), which is addressable by a single URL (e.g., `http://www.site.org/pub/index.html`). A URL has two main components: the symbolic name of the server that houses the object (e.g., `www.site.org`) and the object path name (e.g., `/pub/index.html`). To retrieve all the Web objects composing a Web page, a browser issues multiple requests to the Web server. We refer to a *Web transaction* as the complete interaction that starts when the user sends a request to a Web site and ends when his client receives the last object related to the requested URL. A *session* is a sequence of Web transactions issued by the same user during an entire visit to a Web site.

Summing up, a Web transaction starts when the user makes a request to a Web server, by either typing an URL or clicking on a link, for example `http://www.site.org/pub/index.html`. First, the client extracts the symbolic site name (`www.site.org`) from the requested URL and asks its local domain name server to find out the IP address corresponding to that name. The local name server obtains the IP address by eventually contacting a well-known root of the domain name server hierarchy and ultimately querying the `site`'s authoritative name server. The local name server returns the obtained IP address to the client that establishes a TCP connection with the server or device corresponding to that IP address. After that, the client can send the HTTP request for `/pub/index.html` to the Web server that sends the requested object back. Most Web pages consist of a base HTML file describing the page layout and a number of objects referenced by the HTML file; the page is intended to be rendered to the user as a single unit. Once obtained the base HTML page from the Web server, the client parses it. If the client finds that embedded objects are related to the base page, it sends a separate request for each object. Depending on the HTTP protocol used for the client/server interactions, the subsequent requests can use the same TCP connection (protocol HTTP/1.1) or different TCP connections (protocol HTTP/1.0).

The basic sequence for a Web transaction described above can be altered by several factors, such as caching of the IP address corresponding to the site name at the client browser or at some intermediate name servers, the version of the HTTP protocol being used in the client/server interaction (i.e., support or not for persistent TCP connections), the presence of the requested Web object either in the client local cache or in intermediate proxy servers located on the path between the client and the Web server.

When the Web site is hosted on multiple servers, another alternative is to decide which server of the distributed Web system has to serve a client request. This decision might occur in several places along the request path from the client to the Web site. [33].

It is worth observing that the design and implementation of a Web site architecture consisting of multiple servers is not the only way for improving response time as perceived by the user. Basically, there are two important categories for which we provide just some references for additional reading. They are *external caching* and *outsourcing* solutions.

Probably, the most popular approach for reducing latency time and Web traffic is based on caching of the Web objects that might occur at different levels. An accurate exam of caching solutions would require a dedicated thesis because they are the first techniques proposed in literature and can be deployed at different scales: from server disk caching to proxy caching to browser caching with all possible combinations [112].

In this thesis, we consider some *internal caching* techniques occurring inside the Web-server system, such as Web server accelerators [42, 107] and other techniques for increasing cache hit rate [15, 96], while we exclude external caching solutions, such as proxy servers and cooperative proxies [23, 110], virtual servers (or reverse proxies) [86], Web proxy accelerators [103].

In this thesis, we also exclude solutions where the content provider delegates scalability for its Web services to other organizations. For example, many Web sites contract with third-party Web hosting and co-location providers. Interesting research and implementation issues come from Web-server systems that store and provide access to multiple Web sites [4, 14, 46, 85]. More recently, *Content Delivery Network* (CDN) organizations undertake to serve request traffic for Web sites from caching sites at various Internet borders [1, 2, 67].

1.2 Thesis contributions

The goal of this thesis is to propose and investigate the performance of new algorithms and mechanisms to route the requests in a cluster-based Web system. Our basic premise is the compatibility of all proposed solutions with existing Internet/Web standards and protocols, so that any considered architecture, algorithm and mechanism can be experimented and immediately adopted. The main contributions of this thesis are outlined below.

- We analyze and classify the existing architectures, routing mechanism and dispatching algorithms for locally-distributed Web systems. Our analysis examines and characterizes proposed systems by defining an original taxonomy, evidencing the tradeoffs of the different techniques and pointing out the directions to develop more effective approaches.

- We propose a queuing network model, of a cluster-based Web system. Our model is implemented in a discrete event driven simulator, and is used to evaluate and compare the performance of existing dispatching algorithms and new proposed ones. It is worth to observe that this simulator is limited to this thesis only, but it is built to be a useful performance evaluation tool to investigate the performance of other dispatching algorithms and mechanism for cluster-based Web systems.

- We propose a prototype implementation of a Web cluster called ClubWeb. The prototype implementation is based on commodity of the shelf (COTS) components and Linux operating system. It permits us to analyze the performance of dispatching algorithms and mechanisms in a real testbed environment.

- We propose a new workload model, that integrates the most recent results on Web workload characterization for static requests with dynamic and secure services and a mix of them, as it typically occurs in a modern Web site scenario. We implement the proposed workload both as an input for the simulator and as a benchmark tool used to test the performance of the ClubWeb prototype.

- We propose a new class of content-aware dispatching policies, named CAP. We demonstrate, through simulation and experimental results, that these policies improve the performance of other dispatching algorithms especially for highly heterogeneous workload scenarios.

1.3 Thesis outline

In Chapter 2, we discuss and classify the architecture consisting of multiple nodes distributed in a local area. Our taxonomy is based on the types of information used to manage the client requests and on the mechanisms used to route requests in multi server systems.

In Chapter 3, we present a detailed description of all mechanisms that can be used to route requests in a cluster-based Web system.

In Chapter 4, we first propose a classification of the politics that can be used to select the target server node in a cluster-based Web system, then we give a description of existing and new dispatching algorithms, most of which will be simulated and implemented.

In Chapter 5, we propose a simulation model of a cluster-based Web system, that is based on queuing networks. The simulation model is implemented as a discrete event simulator tool (named *WebSim*), of which we describe the architecture and functionality of the software modules.

In Chapter 6, we describe the implemented Web cluster prototype, called *ClubWeb*.

In Chapter 7, we propose the workload model used in the simulation and prototype experiments. We characterize the behavior of Web users, and the workload to a Web site that provides different Web based services, such as publishing, secure and insecure transaction, e-commerce operations. The workload model has also been implemented as a benchmark tool, used to stress the ClubWeb prototype.

In Chapter 8, we present the results of the performance studies about content-blind dispatching algorithms through the simulation tool and the ClubWeb prototype.

In Chapter 9, we describe a new class of content-aware dispatching algorithms, called *CAP* policies. We discuss the performance results of CAP and others content-aware dispatching algorithms, again through the simulation tool and the ClubWeb prototype.

In Chapter 10, we present the results of the performance studies about dispatching algorithms in a multi-tier clustered Web system. We focus our attention on the second-level dispatching carried out by the Web servers to the back-end nodes.

In Chapter 11, we present our conclusions and topics for future work.

Chapter 2

A taxonomy of locally distributed Web systems

The basic premise of the proposed taxonomy is the compatibility of all analyzed solutions with existing Web standards and protocols so that any considered architecture, algorithm and mechanism could be immediately adopted without any limiting assumption.

Distributed architectures can be differentiated depending on whether or not the name virtualization is extended at the IP level. Given a set of server nodes that hosts a Web site at a single location, we can identify two main classes of architectures:

- **Cluster-based Web System** (or **Web clusters**) where the server nodes mask their IP addresses to clients. The only client-visible address is a *Virtual IP* (VIP) address corresponding to one device which is located in front of the set of servers.

- **Distributed Web System** where the IP addresses of the Web server nodes are visible to client applications. Unlike the cluster-based Web system, this architecture does not have a front-end Web switch, so the client request assignment to a target Web server is typically carried out during the address resolution of the Web site name (*lookup phase*) by the DNS mechanism.
  The study of distributed Web system is outside the scope of this thesis. Major details can be found in [51, 50, 37, 34, 35, 36].

Choosing among visible or masked server IP addresses is a key feature, because each solution implies quite different mechanisms and algorithms for distributing the client requests among the server nodes. In particular, the distributed Web system architecture is the oldest solution, where the request routing is decided by the DNS system with the possible integration of some other naming entity. The cluster-based Web system architecture is a more recent solution where request routing is entirely carried out by the internal components of the Web cluster. We can anticipate that the cluster systems are preferable for a locally distributed Web site because they can provide fine-grained control on request assignment and better availability and security. On the other hand an "IP visible" architecture is more suitable to geographically dispersed Web systems.

The main components of a typical multi-node Web system include a *request routing mechanism* to direct the client request to a target server node, and a *dispatching algorithm* to select the Web server node that is considered best suited to respond.

2.1 Cluster-based Web systems

A cluster-based Web system (briefly, *Web cluster*) refers to a collection of server machines that are housed together in a single location, are interconnected through a high-speed network, and present a single system image to the outside. Each server node of the cluster may be either a workstation, a PC or a symmetric multiprocessor; it usually contains its own disk and a complete operating system. Cluster nodes work collectively as a single computing resource. Massive parallel processing systems where each node satisfies all previous characteristics (e.g., SP-2) can be assimilated to a cluster-based Web system.

In literature some alternative terminology is used to refer to a Web cluster architecture. One common term is also *Web farm*, meaning the collection of all the servers, applications, and data at a particular site [58]. We prefer the term Web cluster, as Web farm is often used to denote an architecture for Web site hosting or co-location.

Although a Web cluster may consist of tens of nodes, it is publicized with one site name (e.g., `www.site.org`) and one virtual IP (VIP) address (e.g., 144.55.62.18). Thus, the authoritative DNS server for the Web site always performs a one-to-one mapping by translating the site name into the VIP address, which corresponds to the IP address of a dedicated front-end node(s). It interfaces the rest of the Web cluster nodes with the Internet, thus making the distribute nature of the site architecture completely transparent to both the user and the client application. The front-end node, hereafter called *Web switch*, receives all inbound packets sent by clients to the VIP address, and routes them to some Web server node. It thus acts as the centralized dispatcher of a distributed system with fine-grained control on client requests assignments.

A high-level view of a basic Web cluster comprising the Web switch, server nodes and back-end nodes is shown in Figure 2.1. It is to be noted that the response line does not appear here because the two main alternatives will be introduced in Section 2.2 when a detailed description will be given in Chapter 3.

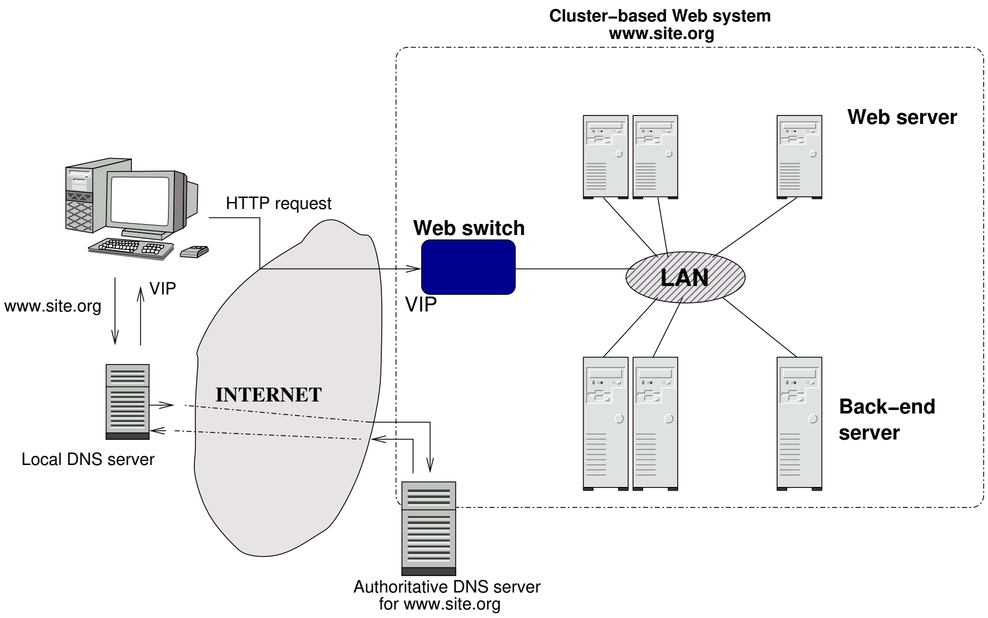


Figure 2.1: Architecture of a *cluster-based Web system.*

The Web switch can be implemented on either special-purpose hardware devices plugged into the network or software modules running on a special-purpose or commodity operating system.

In this thesis, we use the term Web switch to refer to the dispatching entity in general. This definition does not imply that the Web switch is a hardware device that forwards frames based on link layer addresses, or packets based on layer-3 and layer-4 information. Moreover, we prefer not to call the Web switch through the functionality it implements, for example *network/server load balancer*, as it can be found in some literature.

2.2 Request routing mechanisms in Web clusters

The Web switch is able to identify uniquely each node in the system through a private address that can be at different protocol levels, depending on the architecture. More specifically, the server private address may correspond to either an IP address or a lower-layer (MAC) address. There are various techniques to deploy Web clusters, however the key role is always played by the Web switch. For that reason, we first classify the Web cluster architecture alternatives according to the OSI protocol stack layer at which the Web switch routes inbound packets to the target server, that is *layer-4* or *layer-7* Web switches.

The choice of the routing mechanism has also a big impact on dispatching policies because the kind of information available at the Web switch is quite different.

- Layer-4 Web switches perform *content-blind routing* (also referred to as *immediate binding*), because they determine the target server when the client asks for establishing a TCP/IP connection, upon the arrival of the first TCP SYN packet at the Web switch. As the client packets do not reach the application level, the routing mechanism is efficient but the dispatching policies are unaware of the content of the client request.

- Layer-7 Web switches can execute *content-aware routing* (also referred to as *delayed binding*). The switch first establishes a complete TCP connection with the client, examines the HTTP request at application level and then relays it to the target server. This routing mechanism is much less efficient, but it can support more sophisticated dispatching policies. (We refer to layer-7 Web switches according to the ISO/OSI protocol layers, where the application layer is the seventh. Other authors refer to switches that perform content-aware routing as *layer-5* or *application-layer* switches following the TCP model.)

Web cluster architectures based on layer-4 and layer-7 Web switches can be further classified on the basis of the data flow between the client and the target server, the main difference being in the return path server-to-client. Indeed, all client requests necessarily have to flow through the Web switch. On the other hand, the target server can either respond directly to the client (namely, *one-way* architectures) or return its response to the Web switch, that in its turn sends the response back to the client (namely, *two-way* architectures). Figure 2.2 summarizes the taxonomy for Web clusters that we have examined so far. Typically, *one-way* architectures are more complex and more efficient because the Web switch processes only inbound packets, while the opposite is true for two-way architectures because the Web switch has to process both inbound and outbound packets.

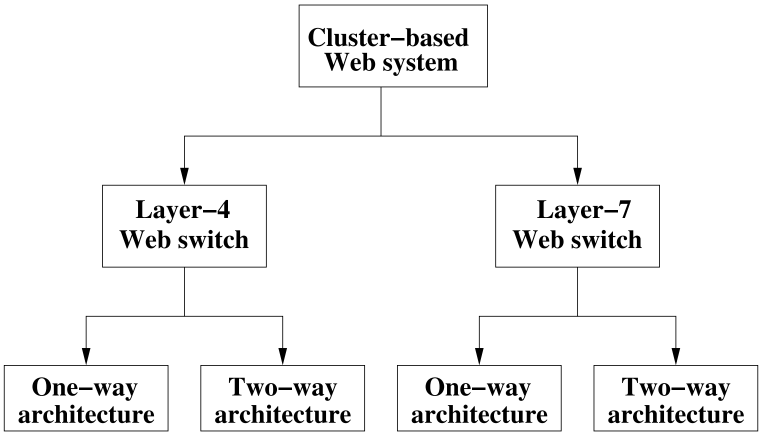


Figure 2.2: Taxonomy of cluster-based architectures.

2.2.1 Solutions based on layer-4 switches

Layer-4 Web switches work at TCP/IP level. Since packets pertaining to the same TCP connection must be assigned to the same Web server node, the client assignment is managed at the TCP session level. The Web switch maintains a *binding table* to associate each client TCP session with the target server.

Layer-4 Web clusters can be classified on the basis of the mechanism used to route inbound packets to the target server and outbound packets to the client. The main difference is in the return way that is, server-to-client. In *two-way* architectures both inbound and outbound packets pass through the Web switch, in *one-way* architectures only inbound packets flow through the Web switch.

Two-way architectures In *two-way* architectures, each server in the cluster is configured with a unique IP address that is, the private address is at IP level. Both inbound and outbound packets are rewritten at TCP/IP level by the Web switch, as shown in Figure 2.3.

One-way architectures In *one-way* architectures inbound packets pass through the Web switch, while outbound packets flow directly from the servers. This requires

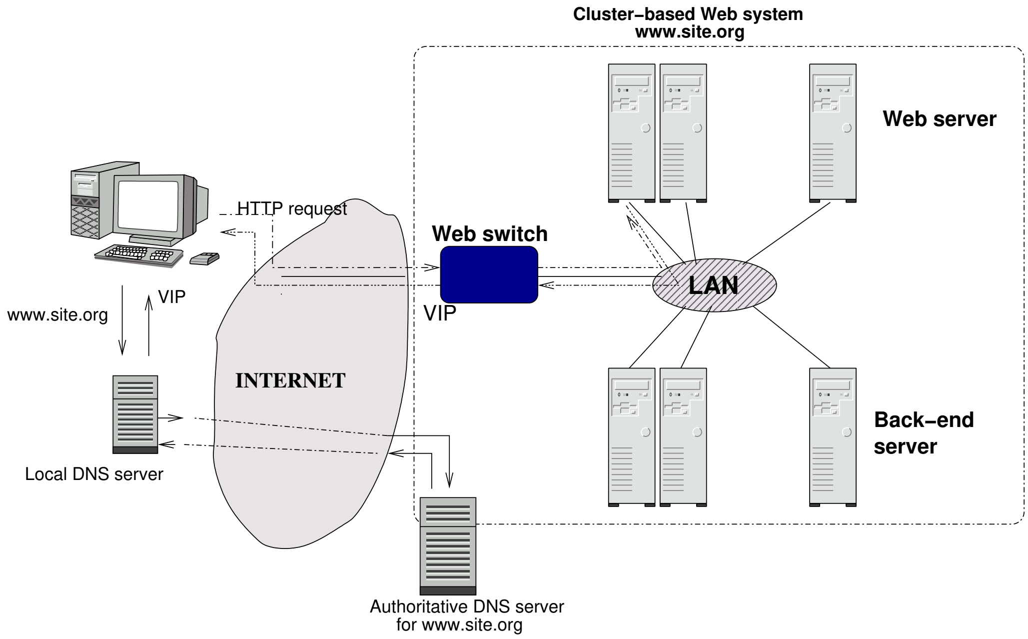


Figure 2.3: Layer-4 two-way architecture.

a separate high-bandwidth network connection for outbound packets. Figure 2.4 shows an example where the outbound packets flow back to the client from the Web server 1. In this Figure, the level of the server private address is intentionally not specified as it can be either at IP level (layer-3) or MAC level (layer-2).

Routing to the target server can be done by means of several mechanisms, such as rewriting the IP destination address and recalculating the TCP/IP checksum of the inbound packet (*packet rewriting*), encapsulating each packet within another packet (*packet tunneling*), forwarding the packet at MAC level (*packet forwarding*). We will describe how each mechanism works in Chapter 3.

2.2.2 Solutions based on layer-7 switches

Layer-7 Web switches work at application level, thus allowing content-aware request distribution. The mechanisms for layer-7 routing should be able to inspect the HTTP request before any dispatching decision. To this purpose, the Web switch must first establish a TCP connection with the client (i.e., the three-way handshake for the TCP connection setup phase must be completed between the client and the Web switch) and then receive the HTTP request at the application level.

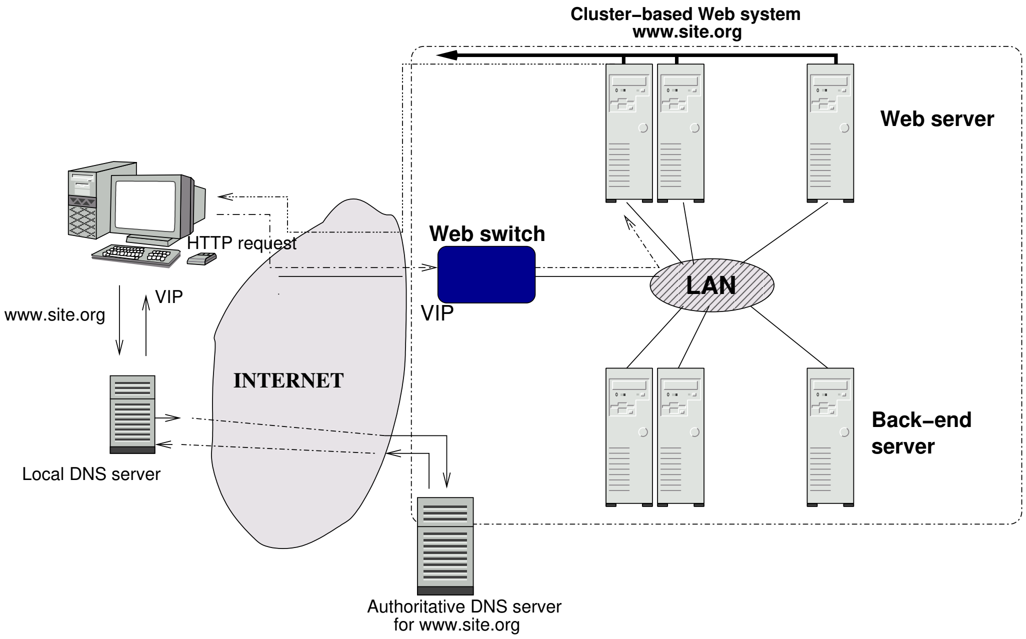


Figure 2.4: Layer-4 one-way architecture.

Similarly to layer-4 solutions, Web cluster architectures based on a layer-7 Web switch can be further classified on the basis of the mechanism used to send outbound packets from server to client. If we consider the data flow through the Web switch, we can distinguish among *one-way* architectures and *two-way* architectures.

Two-way architectures In *two-way* architectures outbound traffic must pass back through the Web switch, as shown in Figure 2.5.

One-way architectures In *one-way* architectures server nodes return outbound packets directly to clients, without passing through the Web switch, as illustrated in Figure 2.6. The mechanism to realize this architecture are more complex, but more efficient than mentioned above.

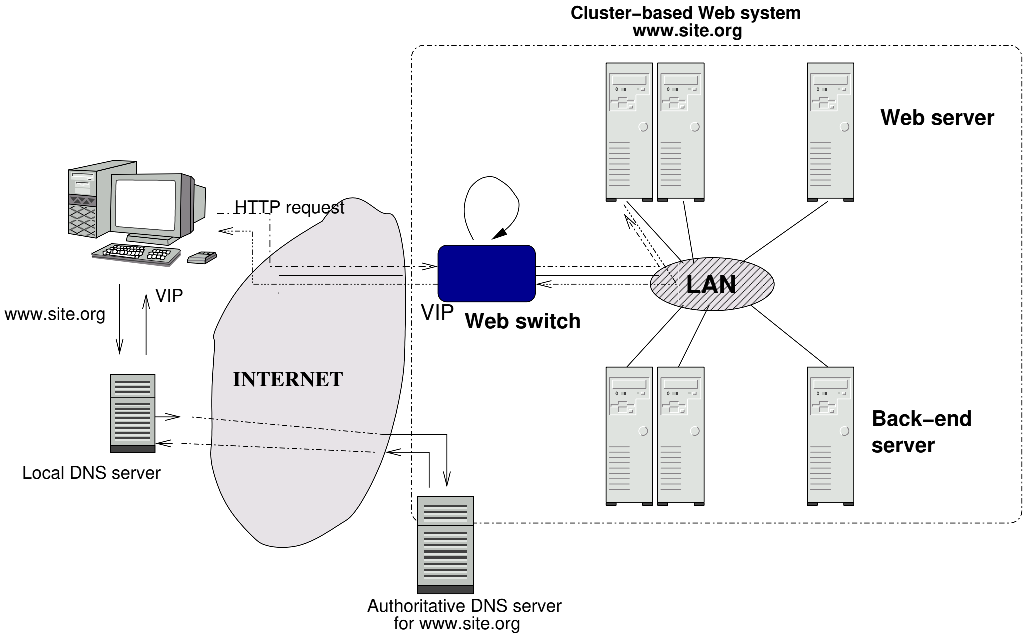


Figure 2.5: Layer-7 two-way architecture.

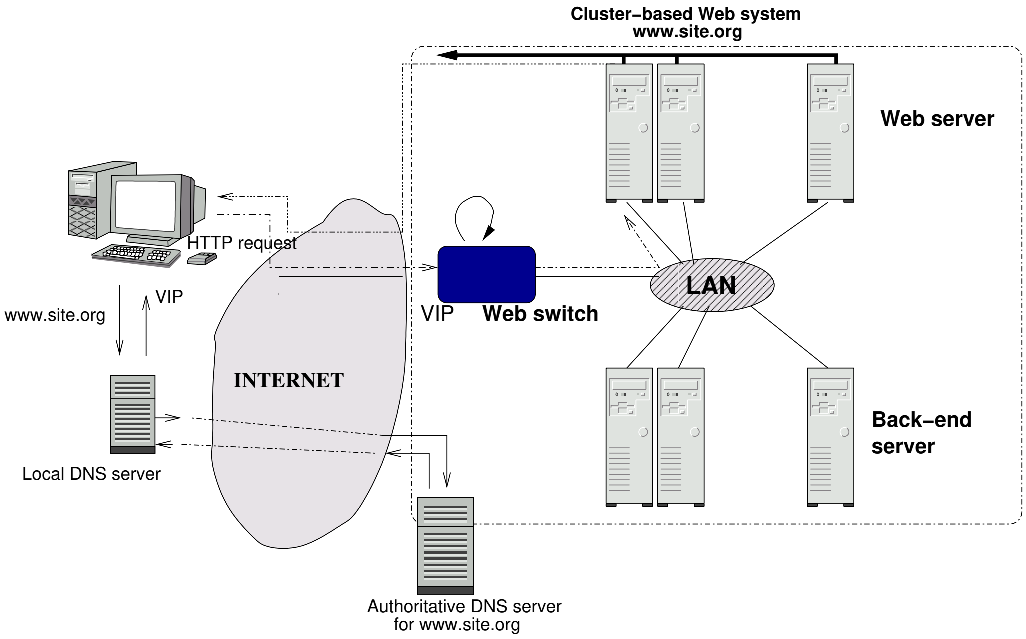


Figure 2.6: Layer-7 one-way architecture.

Chapter 3

Request routing mechanisms

In the previous chapter we presented a taxonomy of locally distributed Web systems. We first classify Web systems in two main categories, according to the OSI protocol stack layer at which the Web switch routes inbound packets to the target server that is, *layer-4* or *layer-7* Web switches. Once decided at which network layer the Web switch works, a more fine classification could be done on the basis of the data flow between the client and the target server. The main difference is in the return way server-to-client. Indeed, all client requests necessarily have to flow through the Web switch. On the other hand, the target server can either respond directly to the client (namely, *one-way* architectures) or return its response to the Web switch, that in its turn sends the response back to the client (*two-way* architectures).

In the next sections we will give a detailed description of the return mechanism for HTTP request, following the taxonomy described in Chapter 2.

3.1 Solutions based on layer-4 switches

Layer-4 Web switches work at TCP/IP level. Since packets pertaining to the same TCP connection must be assigned to the same Web server node, the client assignment is managed at TCP session level. The Web switch maintains a *binding table* to associate each client TCP session with the target server.

Upon receiving an inbound packet, the Web switch examines its header and determines, on the basis of the flag field, whether the packet pertains to a new connection, a currently established one, or none of them. If the inbound packet is for a new connection (i.e., the SYN flag bit is set), the Web switch selects a target server through the dispatching policy, records the connection-to-server mapping in an entry of the binding table, and routes the packet to the selected server. If the inbound packet is not for a new connection, the Web switch looks up the binding table to verify whether the packet belongs or not to an existing connection. If it does, the Web switch routes the packet to the server that is in charge for that connection. Otherwise, the Web switch drops the packet.

To improve Web switch performance, the binding table is typically kept in memory and accessed through a hash function. Each entry contains the tuple `<IP source address, source port, IP destination address, destination port>`, and other information (e.g., beginning time) that may be relevant for some dispatching algorithm.

Layer-4 Web clusters can be classified on the basis of the mechanism used to route inbound packets to the target server and outbound packets to the client. The main difference is in the return way that is, server-to-client. In *two-way* architectures both inbound and outbound packets pass through the Web switch, in *one-way* architectures only inbound packets flow through the Web switch.

3.1.1 Two-way architectures

In *two-way* architectures, each server in the cluster is configured with a unique IP address that is, the private address is at IP level. Both inbound and outbound packets are rewritten at TCP/IP level by the Web switch, as shown in Figure 3.1. Packet rewriting is based on the IP Network Address Translation approach [62].

The Web switch rewrites inbound packets by changing the VIP address to the IP address of the target server in the destination address field of the packet header. Outbound packets from the servers to clients must also pass back through the switch. As the source address in the outbound packets is the address of the server that has served the request, the Web switch needs to rewrite the server IP address with the VIP address, so as not to confuse the client. Furthermore, the Web switch has to recalculate the IP and TCP header checksums for both packet flows.
Let us consider a realistic example. We have a Web cluster composed of three server nodes, as shown in Figure 3.1. The VIP is 160.46.6.217, the internal IP address (we chose the disconnected network 192.168.0.0/255.255.0.0) of the Web switch is 192.168.0.1 and the Web servers have internal IP addresses 192.168.0.2/3/4. Let us suppose that the incoming packet for a new connection has the couple (160.80.85.38, 160.80.6.217) as (source, destination) addresses. The Web switch selects a Web server, for example the Web server with IP address 192.168.0.2, and add a new entry into the

connection table. The next step is to rewrite the destination address of the incoming packet with the IP address of the Web server (192.168.0.2), and to forward it. When an incoming packet belongs to an established connection, the Web switch extracts the IP address of the designed Web server from the connection table, rewrite the header and forward the packet.

The Web server, after receiving all packets of the Web request, builds the http response and send it to the client (indeed the source/destination address of the response packets are (192.168.0.2, 160.80.85.38). The packets are routed to the Web switch which overwrites the source address (192.168.0.2) with the VIP address. Then the outgoing Web cluster packet has the (source,destination) address (160.80.6.217,160.80.85.38), as expected by the client, which is unaware of the distributed nature of the Web system.

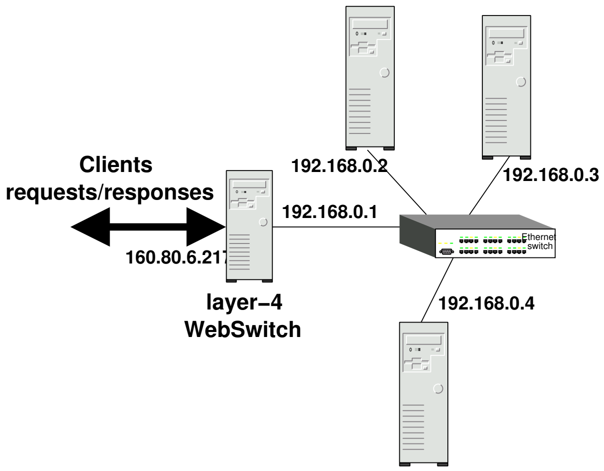


Figure 3.1: Layer-4 two-way architecture.

3.1.2 One-way architectures

In *one-way* architectures inbound packets pass through the Web switch, while outbound packets flow directly from the servers. This requires a separate high-bandwidth network connection for outbound packets. Figure 3.2 shows an example where the outbound packets flow back to the client from the Web server with real IP address RIP1. In this Figure, the level of the server private address is intentionally not specified as it can be either at IP level (layer-3) or MAC level (layer-2).

Routing to the target server can be done by means of several mechanisms, such as rewriting the IP destination address and recalculating the TCP/IP checksum of the

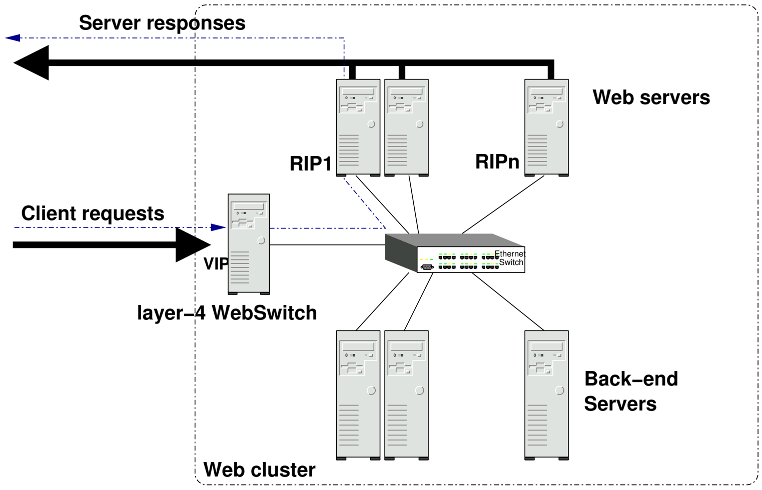


Figure 3.2: Layer-4 one-way architecture.

inbound packet (*packet rewriting*), encapsulating each packet within another packet (*packet tunneling*), forwarding the packet at MAC level (*packet forwarding*). Let us describe how each mechanism works.

Packet single-rewriting. The routing to the target server is achieved by rewriting the destination IP address of each inbound packet: the Web switch replaces its VIP address with the IP address of the selected server and recalculates the IP and TCP header checksum. Thus, the private addresses of the server nodes are at IP level.

The difference from two-way architectures is in the modification of the source address of outbound packets. The Web server, before sending the response packets to the client, replaces its IP address with the VIP address and recalculates the IP and TCP header checksum [59]. To differentiate packet rewriting operations, we call *double-rewriting* those performed by two-way architectures and *single-rewriting* those carried out by one-way architectures.

Packet tunneling. IP tunneling (or IP encapsulation) is a technique to encapsulate IP datagrams within IP datagrams, thus allowing datagrams destined to one IP address to be wrapped and redirected to another IP address [99]. The effect of IP tunneling is to transform the old headers and data into the payload of the new packet. The Web switch tunnels the inbound packet to the target server by encapsulating it within an IP datagram. The header of this datagram contains the VIP address and the server IP address as source and destination address, respectively. The server private addresses are at IP level as in packet rewriting. This mechanism requires that all servers support IP tunneling and have one of their tunnel devices configured with the VIP address. When the target server receives the encapsulated packet, it strips the IP header off and finds that the inside packet is destined to the VIP address configured on its tunnel device. Then, the server processes the request and returns the response directly to the client. Figure 3.3 shows the packets flow. The Web switch receives an incoming packet(1), selects a Web server, encapsulates the packet in a tunneling IP packet (2) and forwards it to the Web server (3) through a local or wide area network. The designed Web server receives the packet and decapsulate it. After the client request is processed, the response is produced and sent directly to the client(4).

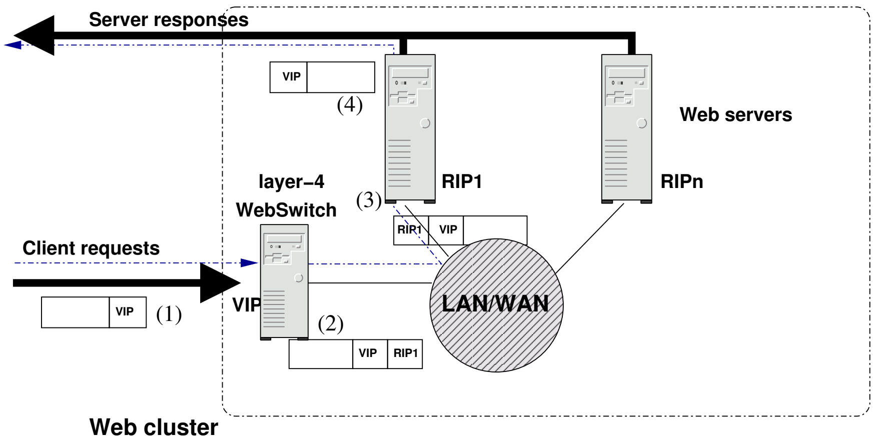


Figure 3.3: Packet flow in the IP tunneling mechanism.

Packet forwarding. This approach assumes that the Web switch and the server nodes are on the same local network. More specifically, the switch and the servers must have one of their network interfaces physically linked by an uninterrupted LAN segment.

The virtual IP address is shared by the Web switch and all of the servers in the cluster through the use of primary and secondary IP addresses, i.e. each server is configured with the VIP address as its secondary address. This may be done through the use of loopback interface aliasing, for example by using the `ifconfig` Unix command. Even if all nodes share the VIP address, the inbound packets reach the Web switch because the server nodes have disabled the Address Resolution Protocol (ARP) mechanism (otherwise, a collision would occur). Hence, the Web switch can forward the inbound packet to the target server by using its physical address on the LAN (i.e., the MAC address) without modifying the TCP/IP header. The packet forwarding is achieved by letting the Web switch rewrite the layer-2 destination address to the MAC address of the server and retransmitting the frame on the network. For this reason, packet forwarding is also referred to as *MAC address translation*. Unlike packet single-rewriting and packet tunneling mechanisms, the server private addresses are now at MAC level.

When the server receives the forwarded packet, it processes it as a packet destined for itself, since it shares the VIP address. Then, it returns the response directly to the client.

Figure 3.4 shows the packets flow. When the first packet of a connection arrives, the Web switch select a Web server and update the connection table (1), than forward the packet to the designed server (2). All the other packet of this connection are directly forwarded to the designed server on the basis of MAC address stored on the connection table. The Web server receives packets as directly coming from the client (3), and knows the client IP address to which it can send directly the response (4).

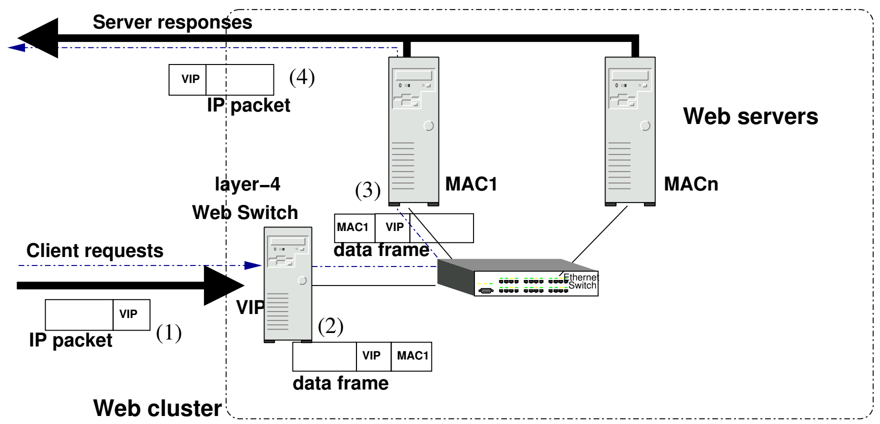


Figure 3.4: Packet flow in the IP tunneling mechanism.

3.1.3 Products based on a layer-4 Web switch

Table 3.1 classifies some commercial products and research prototypes that work at the TCP/IP layer. Some products, such as Linux Virtual Server [82], appear in multiple table entries because they can be configured to support more than one request routing mechanism.

Two-way	One-way		
Packet double-rewriting	*Packet single-rewriting*	*Packet tunneling*	*Packet forwarding*
Cisco's LocalDirector [47]	TCP Router [59]	Linux Virtual Server [82]	IBM Network Dispatcher [73, 74]
Magicrouter [5]			Linux Virtual Server [82]
Linux Virtual Server [82]			F5 Networks' BIG-IP [63]
LSNAT [108]			LSMAC [68]
F5 Networks' BIG-IP [63]			Intel's NetStructure Traffic Director [75]
Foundry Networks' ServerIron [65]			Nortel Networks' Alteon 180 [93]
IBM Network Dispatcher [74]			Radware's WSD Pro [101]
Coyote Point's Equalizer [52]			Foundry Networks' ServerIron [65]
Allot's NetBalancer [3]			ONE-IP [55]

Table 3.1: Layer-4 Web clusters.

We compare some prototypes and products based on a layer-4 Web switch by first considering two-way architectures and then one-way architectures.

Two-way solutions

Layer-4 Web switches based on double-packet rewriting are often implemented on specialized hardware because of their relative simplicity. Examples are LocalDirector from Cisco Systems [47] and Foundry Networks' ServerIron [65]. However, pure-software implementations are also possible, such as Linux Virtual Server [82] and the Magicrouter prototype from Berkeley [5].

Two-way architectures typically offer various solutions for the dispatching policies implemented at the Web switch, ranging from static algorithms to client and server state aware algorithms. For example, Cisco's LocalDirector [47] is provided with the *least connections* policy that selects the server with the least number of active connections, and the *Fastest Response Time* algorithm that dispatches the request to the server that responded as the fastest to previous connection requests. Magicrouter [5], besides the stateless round-robin and random algorithms, offers also the *incremental load* policy, which is similar to selecting the least loaded server and is based on the current server load plus an adjustment related to the number of active connections.

Two-way architectures (e.g., Cisco's LocalDirector and Foundry Networks' ServerIron [65]) can also support some stateful services, such as SSL, through the use of a sticky flag. This is achieved by directing multiple connections from the same client to the same server within a period of time that is set to 5 minutes by default. However, SSL session reuse provided by layer-4 Web switches may lead to load imbalance among the server nodes because these switches are not able to distinguish among different clients that are behind the same proxy or firewall.

One-way solutions

Most existing products and prototypes based on one-way architectures provide request routing through the *packet forwarding* mechanism. Among the others, we recall the IBM Network Dispatcher [73, 74] which is the load balancing component of the IBM WebSphere Edge Server, and ONE-IP from Bell Labs [55] which is one of the first implementations of a layer-4 Web switch based on packet forwarding. In these systems, the switch forwards packets destined to the Web cluster to a selected server by using its MAC address on the LAN, without modifying the TCP/IP headers. This is possible because all the server nodes share the same VIP address which is bound to the loopback interface (for example through the `ifconfig` command in Unix/Linux systems). It is important to observe that some commercial products define the packet forwarding mechanism through different names. This is the case of F5's nPath [63] and Foundry Networks' SwitchBack [65].

A small subset of solutions use request routing mechanisms that differ from packet forwarding. For example, IBM TCP router [59] implements packet single-rewriting which allows the Web servers to respond directly to clients by modifying outbound packets, while one of the three request routing mechanism supported by Linux Virtual Server [82] is based on IP packet tunneling.

Similarly to two-way architectures, one-way solutions offer many alternatives for the dispatching policies implemented at the Web switch. As explained in Section 4.2.5, the main difference between the two architectures is in the server state information directly available at the Web switch. For example, IBM Network Dispatcher uses a dynamic Weighted Round-Robin algorithm to distribute connections among the server nodes [73]. ONE-IP supports two static dispatching algorithms, called *routing-based dispatching* and *broadcast-based dispatching* [55]. In both solutions, the destination server is selected by applying a hash function that maps the client IP address into a server identifier. The difference between the two solutions lies in the cluster component that applies the hash function. In routing-based dispatching, the Web switch selects the target server using the hash function, while in broadcast-based dispatching the Web switch broadcasts the packets destined to the VIP address to every server in the cluster; each server evaluates whether it is the actual destination of the packets by applying the hash function.

One-way solutions are also able to support stateful services, such as SSL session reuse (although the same limitation of two-way architectures applies). For example, the Network Dispatcher provides a *client affinity* mechanism that is similar to Cisco's sticky flag.

3.2 Solutions based on layer-7 switches

Layer-7 Web switches work at application level, thus allowing content-aware request distribution. The mechanisms for layer-7 routing are more complex than those for content-blind routing, because the HTTP request is inspected before any dispatching decision. To this purpose, the Web switch must first establish a TCP connection with the client (i.e., the three-way handshake for the TCP connection setup phase must be completed between the client and the Web switch) and then receive the HTTP request at the application level. On the other hand, a layer-4 Web switch determines the target server as soon as it receives the initial TCP SYN packet, before the client sends out the HTTP request. Figure 3.5 shows the different instants in which the request decision is made by a Web switch that routes new connections at layer-4 or layer-7.

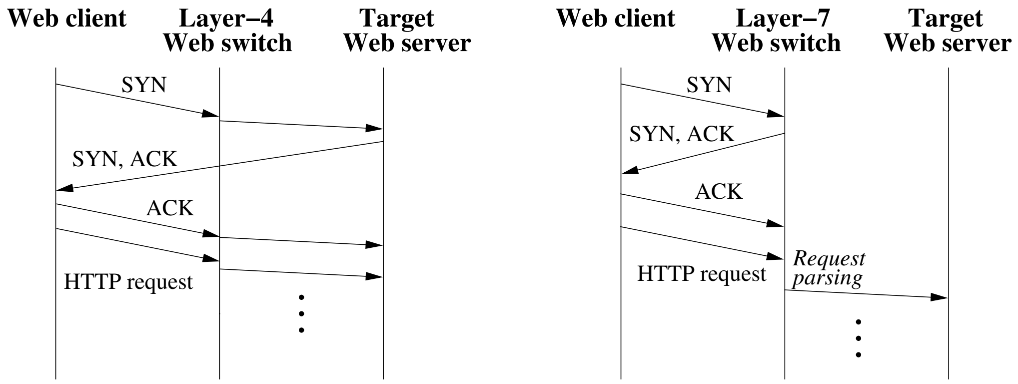


Figure 3.5: Operations of layer-4 routing (left) and layer-7 routing (right).

Similarly to layer-4 solutions, Web cluster architectures based on a layer-7 Web switch can be further classified on the basis of the mechanism used to send outbound packets from server to client. If we consider the data flow through the Web switch, we can distinguish among *one-way* architectures and *two-way* architectures.

3.2.1 Two-way architectures

In *two-way* architectures outbound traffic must pass back through the Web switch, as shown in Figure 3.6.

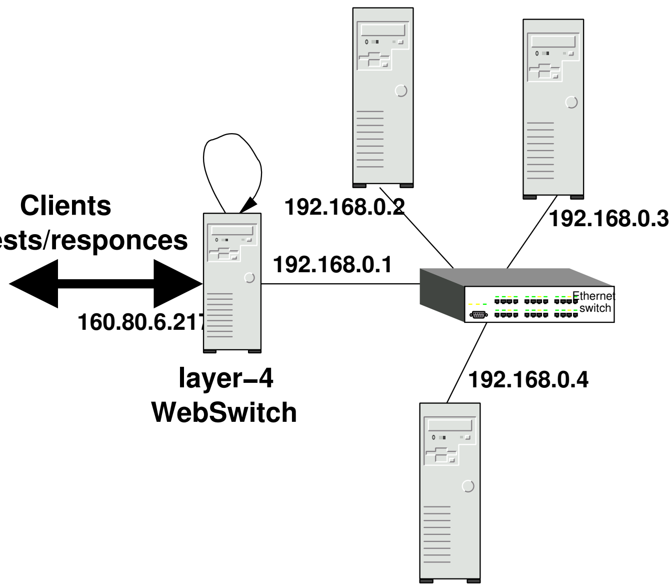


Figure 3.6: Layer-7 two-way architecture.

The proposed approaches basically differ in the mechanism the Web switch uses to route requests to the target server.

TCP gateway. In this architecture, an application level proxy running on the Web switch mediates the communication between the client and the server. This proxy accepts client connections and maintains TCP persistent connections with all the server nodes. When a request arrives on a client connection, the proxy forwards the client request to the target server through the corresponding TCP persistent connection. When the response arrives on the persistent connection back from the server, the Web switch forwards it to the client through the other connection.

TCP splicing. This mechanism aims to improve the TCP gateway approach that is computationally expensive. Now, packet forwarding occurs at network level between the network interface driver and the TCP/IP stack and is carried out directly by the operating system. Cohen et al. present a TCP splicing implementation in Linux operating system [49].

Figure 3.7 outlines the operations involved in TCP splicing. Once the TCP connection between the client and the Web switch has been established and the persistent TCP connection between the switch and the target server has been chosen, the two connections are spliced together. In such a way, IP packets are forwarded from one endpoint to the other without having to cross the TCP layer up to the application layer on the Web switch. Once the client-to-server binding has been determined, the Web switch handles the subsequent packets by changing the IP and TCP packet headers (IP addresses and checksum recalculations), so that both the client and the target server can recognize these packets as destined to them.

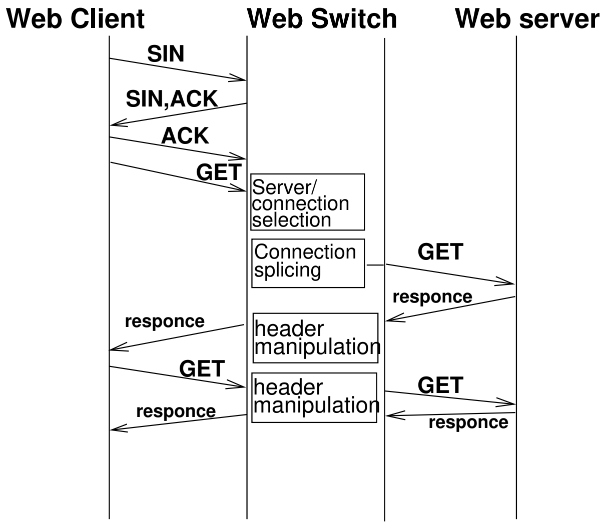


Figure 3.7: TCP splicing operation carried on by the Web switch.

The TCP splicing mechanism can be also realized by hardware-based switches. Apostolopoulos et al. presented an hardware-based Web switches, consisting of programmable port controllers connected to a switch core and a general purpose CPU [11]. TCP splicing is used in this switch to take the CPU out of the data path once the URL is obtained and connection to the server is established. After splicing, all packet processing is handled by the port controller.

3.2.2 One-way architectures

In *one-way* architectures the server nodes return outbound packets directly to clients, without passing through the Web switch, as illustrated in Figure 3.8.

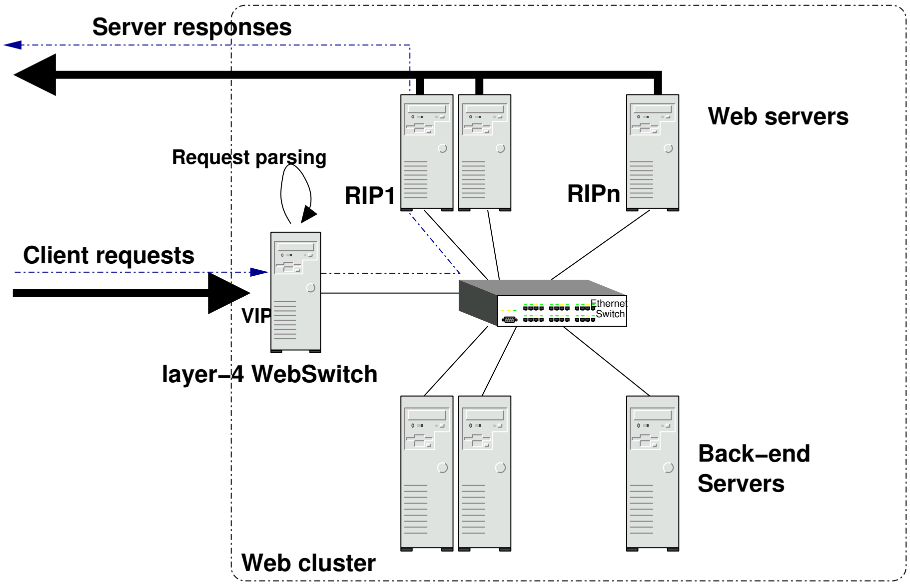


Figure 3.8: Layer-7 one-way architecture.

TCP handoff. Once the Web switch has established the TCP connection with the client and selected the target server, it hands off its endpoint of the TCP connection to the server [96]. The handoff protocol is layered on top of TCP and runs on the Web switch and the servers, thus requiring changes in the operating systems of both components. The TCP handoff mechanism remains transparent to the client, as data sent by the servers appear to be coming from the Web switch. Any acknowledgment packets sent by the client to the switch are forwarded to the target server by a module running at the bottom of the switch protocol stack.

As shown in Figure 3.9, the Web client establishes a TCP connection with the Web site refereeing the VIP. Once received the http request over the previously established TCP connection, the Web switch selects the appropriate Web server and establishes an hand-off connection with it. This means that all new client packets will be forwarded by the Web switch to the Web server, over the hand-off connection, after a layer-2 header handling. The Web server receives the packet as coming directly from the Web client, and it is able to reply to the client without other interactions with the Web switch.

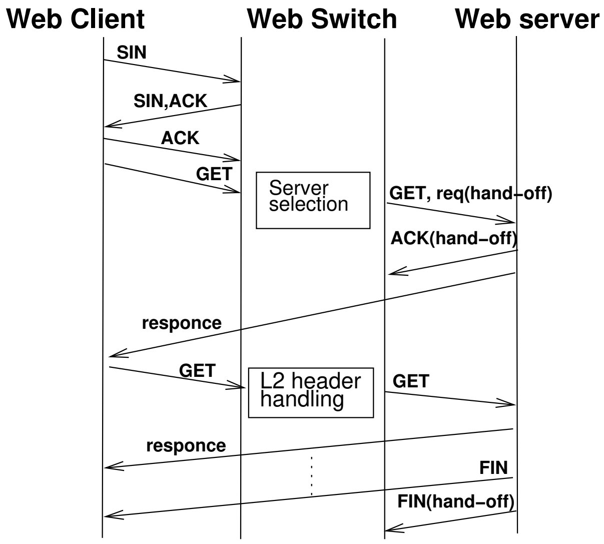


Figure 3.9: TCP hand-off operation carried on by the Web switch.

The handoff mechanism allows to handle HTTP persistent connections by letting the Web switch assign HTTP requests in the same connection to different target servers [15]. There are two mechanisms to support persistent connections: the handoff protocol can be extended by allowing the Web switch to migrate a connection between servers (*multiple handoff*), the first target server forwards the request it cannot serve to a second server that sends the response back to the client (*back-end forwarding*).

TCP connection hop. This is a software-based proprietary solution realized by Resonate [102]. Once the Web switch has established the TCP connection with the client and selected the target server, it hops the TCP connection to the server. This is achieved by encapsulating the IP packet in an RPX packet and sending it to the server [102]. Since the server shares the same VIP address, it can reply directly to the client. Acknowledgment packets and persistent session information from clients are managed by the Web switch.

3.2.3 Products based on a layer-7 Web switch

In Table 3.2 we classify the cluster-based Web systems that work at the application layer. Some products listed herein, such as Foundry Networks' ServerIron [65] and F5 Networks' BIG-IP [63], have already been considered in Table 3.1 as they can be configured to support both layer-4 and layer-7 routing mechanisms.

Two-way		**One-way**	
TCP gateway	*TCP splicing*	*TCP handoff*	*TCP connection hop*
IBM Network Dispatcher CBR [74]	Nortel Networks' Alteon Web OS [93]	ScalaServer [15, 97]	Resonate's Central Dispatch [102]
CluBWeb [39]	Lucent's Web Switch[49, 83]	[109]	
HACC [115]	Cisco's CSS [47]	CluBWeb [6]	
	Foundry Networks' ServerIron [65]		
	F5 Networks' BIG-IP [63]		
	Radware's WSD Pro [101]		
	Zeus's Load Balancer [114]		
	L5 [10]		
	[113]		
	Array Networks' Array 500 [17]		

Table 3.2: Layer-7 Web clusters.

Two-way solutions

Request routing in two-way architectures can be implemented in hardware (e.g., Cisco's CSS [47], L5 [10], Nortel Networks' Alteon Web OS [93], and Radware's WSD Pro [101]), completely in software (e.g., the IBM Network Dispatcher CBR product [74], the Harvard Array of Cheap Computers (HACC) [115] and the CLUster-Based Web (ClubWeb) prototypes [39]), or through a hardware switch combined with some control functions implemented in software (for example, F5 Networks' BIG-IP [63] consists of a hardware box with a modified BSDi-Unix kernel, while Lucent's Web Switch [83] is a Linux-based platform.

The request routing mechanism can be implemented at the application layer by configuring the Web switch to work as an enhanced TCP gateway or reverse proxy (e.g., IBM Network Dispatcher CBR [74], HACC [115], and ClubWeb [39]). An alternative solution is to route client requests at a lower layer by letting the Web switch splice the TCP connection established with the client to a connection with a selected server. Nortel Networks' Alteon Web OS [93], F5's BIG-IP [63], and Lucent's Web Switch [49, 83] use the TCP splicing mechanism. To this purpose, the Web switch modifies the TCP header of every packet that travels between the client and the server to perform TCP/IP header recalculation and sequence number adjustments. In Lucent's Web Switch [83] two components implement the switching functionality on the Web switch. The first component is a user-level proxy that accepts connections from clients, parses the HTTP requests, and selects a target server. The proxy then removes itself from the data path by requesting a kernel module to splice the TCP connections together. To limit the bottleneck risks at the Web switch, the splicing functions can be implemented on specialized hardware. For example, Nortel Networks' Alteon Web OS [93] relies on a dedicated network multi-processor integrated with a parallel-processing operating environment.

As regarding the dispatching algorithms in two-way layer-7 switches, the most sophisticated policies are typically implemented in research prototypes. For example, in the HACC prototype [115] the target server node is selected on the basis of the locality properties and capacity of the nodes. The Web switch monitors the server loads and the state of the files stored by means of a tree-based structure. Requests for new objects are assigned to the least loaded server and the tree-based structure is updated. To improve reference locality in disk cache, subsequent requests for the same object are assigned to the same server. In the ClubWeb prototype [39], the Web switch selects the target server on the basis of the CAP policy described in Section 9.

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. Typically, each commercial product provides a set of content-aware dispatching policies. The companies use different names, but the substance is similar. Let us give some examples. In Nortel Networks' Alteon Web OS [93] the content-aware policy is basically a service partitioning algorithm that allows specialized servers to store specific object types or provide specific services. Hence, the client request is assigned by the Web switch to a target server that is selected on the basis of URL (or URL substring) matches. Foundry Networks' ServerIron [65] supports URL hashing, in which the Web switch examines information in the HTTP request and maps this information to one of the servers. All the HTTP requests that contain the same information are always assigned to the same server. F5 Networks' BIG-IP [63] can use cookie information stored in an HTTP request to direct requests from the same client to the same server. A more sophisticated algorithm that resembles the LARD policy [97] and aims at improving the locality of reference in request streams is implemented on the Zeus's Load Balancer [114].

One-way solutions

One-way solutions at layer-7 are typically software-based implementations that require modifications at the operating system level on both the Web switch and the servers. These intervents can be done in the internal kernel. An alternative is to realize the supported mechanism as a loadable kernel module not requiring modifications to the native kernel. For example, to support the TCP handoff protocol described in Section 3.2 the ScalaServer prototype [15, 97] from Rice University modifies the FreeBSD operating system, while the CluBWeb prototype [6] modifies the Linux kernel. A set of dynamically loadable modules that provide the TCP handoff mechanism in a STREAMS-based TCP/IP implementation has been proposed in [109].

To the best of our knowledge Resonate's Central Dispatch [102] is the only commercial product that implements a one-way architecture with a layer-7 Web switch. This solution requires the installation of a kernel module on both the Web switch and the servers to support the proprietary TCP connection hop mechanism (see Section 3.2).

It is worth noting that all the proposed request routing mechanisms are transparent to the Web server running on the server nodes, thus allowing the use of any off-the-shelf Web server software. Incoming traffic on already established connections is forwarded to the target server through an efficient forwarding module layered between the network interface card and the TCP/IP stack. A distinguishing feature of the request routing mechanism deployed by ScalaServer is the support to HTTP/1.1 persistent connections which allows to assign HTTP requests to different servers through the same TCP connection. This is achieved through a *back-end forwarding* mechanism, that allows the original target node to forward a request to a second server selected by the Web switch [15]. Assigning individual HTTP requests traveling on the same TCP connection to multiple servers appears to be also a feature of Netscaler's Request Switch [92], but no information is currently provided about the routing mechanism because of patent pending reasons.

As regards the server selection, one-way solutions working at layer-7 typically analyze the HTTP header content prior to dispatching the request to an appropriate server. In the ScalaServer [97] and ClubWeb prototype [6] the Web switch selects a server on the basis of the LARD and CAP policy, respectively. Both policies are described in Section 9.

In Resonate's Central Dispatch, the Web dispatching algorithm uses client information in addition to server availability and performance. Specifically, upon receipt of an HTTP request, the Resonate's switch parses the URL to determine the content being requested and applies a configurable scheduling rule to it. If more than one server is available to serve the request, the Web switch transfers the TCP connection to the least loaded server.

3.3 Comparison of routing mechanisms for Web clusters

Figure 3.10 summarizes the taxonomy for Web cluster by providing further details with respect to the taxonomy in Figure 2.2. We first discuss the different solutions for layer-4 and layer-7 routing, then we compare the two classes of routing mechanisms.

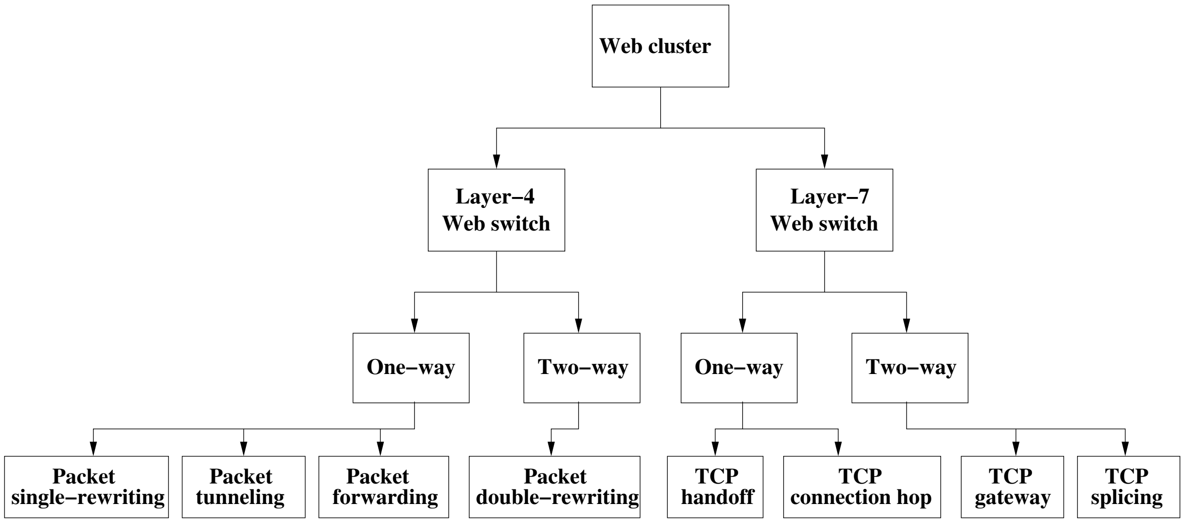


Figure 3.10: Detailed taxonomy of Web cluster architectures.

3.3.1 Layer-4 routing mechanisms

In two-way solutions, the server nodes may be in different LANs, with the only constraint that both inbound and outbound traffic flow through the Web switch. The main problem of these mechanisms is that the Web switch must rewrite inbound as well as outbound packets, and outbound packets typically outnumber inbound packets. Thus, the scalability (in terms of throughput) of Web clusters that use a two-way architecture is limited by the Web switch ability to rewrite packets and recalculate their checksums, even if a dedicated hardware support can be provided for the checksum operations. On the other hand, a layer-4 Web switch that uses a one-way solution can sustain a larger throughput before becoming the system bottleneck. Thus, the system performance is only constrained by the ability of the switch to set up, look up, and tear down entries in the binding table.

If we consider the different mechanisms for one-way architectures, we see that packet single-rewriting causes the same overhead as packet double-rewriting, but it reduces switch operations because the more numerous outbound packets are rewritten by the Web servers and not by the Web switch. However, this requires modifications in the kernel of the server operating system.

Packet forwarding mechanisms aim to limit the overhead of packet rewriting. As the Web switch processes only inbound packets and, beside that, at MAC level that avoids expensive checksum recalculations, the cluster scalability is primarily limited by the capacity of the cluster network link to Internet. The drawback of packet forwarding is that it requires the same network segment to connect the Web switch and all the Web server nodes. However, this restriction has a very limited practical impact since the Web switch and the servers are likely to be connected through the same LAN.

Solutions that employ packet tunneling have good scalability, although lower than packet forwarding. Moreover, they require the servers to support IP tunneling which is not yet a standard for current operating systems.

3.3.2 Layer-7 routing mechanisms

An advantage of two-way architectures combined with a layer-7 Web switch is that caching can be implemented on the Web switch. This allows such device to reply directly to a request if it can be satisfied from the cache with a consequent load decrease on server nodes.

A simple realization is the main advantage of the TCP-gateway approach that can be implemented on any operating system. The main drawback is the cost of this solution as both request and response data flows through the Web switch up to the application level. TCP splicing reduces TCP gateway overhead, because it eliminates the expensive copying and context switching operations that result from the use of an application-level proxy. However, even in this instance, the Web switch can easily become the bottleneck of the cluster as it has to modify the TCP/IP headers.

Layer-7 Web switches that use one-way solutions enable the server nodes to respond directly to the clients, thus offering higher scalability than two-way solutions. In particular, the TCP handoff approach scales better than TCP splicing as shown in [16]. The main drawback of one-way solutions at layer-7 lies in that they require modifications in the operating system of both the Web switch and the servers.

3.3.3 Layer-4 vs. layer-7 routing

The main advantage of layer-7 routing mechanisms over layer-4 solutions is the possibility of using content-aware dispatching algorithms at the Web switch. We will see in Section 9 that through these policies it is possible to achieve high disk cache hit rates, partition the Web content among the servers, employ specialized server nodes, assign subsequent SSL sessions to the same server, and achieve a fine grain even with HTTP/1.1 persistent connections.

On the other hand, layer-7 routing mechanisms introduce severe processing overhead at the Web switch to the extent that may cause the dispatcher to limit seriously Web cluster scalability [16, 107]. As an example, Aron et al. show in [16] that the peak throughput achieved by a layer-7 switch that employs TCP handoff 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. Many solution have been proposed to overcome this drawback combining content-blind and content-aware request distribution; we discuss some of this solution in Section 3.4.

Table 3.3 outlines the main features and tradeoffs of the various mechanisms we have discussed for Web cluster architectures.

3.4 Integrated dispatching mechanisms

Reliability, scalability and security are critical qualities in Web-based information services. Indeed, one common objection raised against the Web cluster architecture concerns the Web switch that represents a single point of failure and a potential system bottleneck. The reliability issue can be addressed by a replication of the switch device integrated with some heartbeat technique (e.g., [84]) or by some distributed dispatching mechanism (e.g., [18]). This section focuses on the scalability issues and presents some extensions to the basic Web cluster architecture described in Sections 3.1 and 3.2 . The most interesting proposals for improving scalability

Approach	*Routing*	*Data flow*	*Pros*	*Cons*
Layer-4 (two-way)	Content-blind	Inbound/outbound	Flexible, portable	Switch throughput, TCP connection grain control
Layer-4 (one-way)	Content-blind	Inbound	Simple, fast routing	Network topology, TCP connection grain control
Layer-7 (two-way)	Content-aware	Inbound/outbound	Simple, caching, server specialization, HTTP grain control, content partition, SSL session reuse	Switch bottleneck, slowest routing
Layer-7 (one-way)	Content-aware	Inbound	Server specialization, HTTP grain control, content partition, SSL session reuse	Switch bottleneck, complex routing

Table 3.3: A summary of local routing mechanisms for Web clusters.

combine request routing mechanisms and exploit some system caching techniques.

A simple solution to avoid a system bottleneck at the Web switch is to use multiple Web clusters, each with a front-end switch and a visible IP address. During the address resolution phase, the authoritative DNS can dispatch the client requests among the Web clusters through simple static algorithms, such as round-robin. Each Web switch can use another dispatching algorithm to share the load among the Web servers of each cluster. A similar architecture was first proposed by Dias et al. where the Web switch operated at the TCP layer [59], and it is now adopted by several geographically distributed Web systems. Indeed, when all strategies have been implemented, and the Web cluster scalability and reliability is primarily limited by the network connection to Internet, the best alternative for a content provider that does not want to refer to outsourcing solutions is to distribute Web clusters over different Internet zones.

An interesting idea for improving Web cluster scalability is to combine the performance of a content-blind dispatcher (DNS, layer-4 Web switch) with the caching features of a content-aware dispatcher, that can be implemented by a layer-7 switch or by a Web server.

A change of the basic Web cluster architecture integrates a layer-4 Web switch with two or more layer-7 Web switches that take content-aware dispatching decisions and provide some caching functionality. Indeed, the caches store frequently accessed Web objects and respond to requests for these objects, thus relieving that workload from the Web servers. The first proposal where caching is integrated into one layer-7 Web switch is in [81], the so called *Web server accelerator*. A first improvement was to use sophisticated mechanisms that allow caching of dynamic objects too [41]. An evolution of the basic Web server accelerator architecture adds a set of Web server accelerators between the Web switch and the Web servers [107, 106], basically an array of caches with some dispatching functionality. The goal is to improve Web cluster performance by increasing cache hit rates. The front-end switch can be a layer-4 or a layer-7 Web switch. When a layer-4 switch receives the requests, it routes them to the cache nodes regardless of URL contents. Therefore, the request may be sent to a wrong cache node that does not contain a cached copy of the object. If that happens when the requested object is small, the first node gets the requested object from a node containing the cached copy and sends the response back to the client. Instead, when the requested object is large, the first node hands off the TCP connection to a node containing a cached copy of it, and this node responds directly to the client without passing through the first node or the layer-4 switch. The use of a layer-7 Web switch as a front-end can reduce the work of the cache nodes and limit the percentage of request redirection, however it achieves a lower aggregate throughput because of the overhead of content-aware routing mechanisms.

A different approach to improve Web cluster efficiency is to perform content-aware dispatching or caching through the Web servers instead of additional layer-7 Web switches. The first dispatching level carried out by a layer-4 Web switch (or by the authoritative DNS if we refer to a distributed Web system) selects one Web server typically by means of a static algorithm. There are various proposals in this sense, that basically differ for the way the Web content is distributed (that is, replicated [16, 38] or partitioned [44]) and the system information is shared among the nodes.

If each node can access all Web documents, the server selected by the Web switch may redirect the request for load sharing or caching reasons, for example it is aware that another server is lightly loaded or holds the requested object in the disk cache. This system information may be centralized in a dispatcher component [16] or distributed among all the servers [44, 38].

In the prototype proposed by Aron et al. [16] the layer-4 Web switch selects one server through the dynamic Weighted Round-Robin algorithm. The server accepts the connection, parses the client request, and contacts a central dispatcher located in the internal LAN that decides about (re-)assignment on the basis of the LARD policy. If the dispatcher selects a different server, the first contacted server hands off the connection towards the other server, that responds directly to the client. When a TCP connection handoff occurs, the server sends a message to the layer-4 switch by instructing it to route packets not to the original server but to the node chosen by the dispatcher. In this architecture the second-level dispatching decision is centralized, while the second-level routing is distributed and carried out by each server. The motivation is that the processing overhead on a layer-7 switch is caused by the routing mechanism and not by the server selection task. Although the authors demonstrate that the dispatcher is not the system bottleneck [16], a distributed dispatching algorithm carried out by each server node could avoid the communication overheads with the centralized dispatcher. This alternative is chosen by [38] that implement a Web cluster with a global caching mechanism. Depending on the size of the requested object, and its presence in the local cache or in the caches of other servers, the first contacted Web server replies to the client immediately or after having retrieved the file from another server. Server load and caching information are periodically broadcasted by each server.

In the prototype proposed by [44] the Web content is not entirely replicated. Just a small set of the most popular files (namely, *core*) can be accessed by any server, while the other files are partitioned among the nodes. If the client request is for a core file, the request is processed by the first contacted server, otherwise the server redirects it to the designated server node. The global information about document location is stable, because the core is determined by analyzing periodically (e.g., daily) the workload access patterns.

Chapter 4

Dispatching algorithms

After the analysis of the mechanisms for routing requests, in this Chapter we propose the policies that can be used to select the target server node in a cluster-based Web system.

In a Web cluster the dispatching policy is carried out by the Web switch that acts as a global scheduler for the system[1]. Global scheduling policies have been classified in several ways, following different criteria [40, 104, 105, 111]. There are several alternatives, such as *load balancing* vs. *load sharing*, *centralized* vs. *distributed*, and *static* vs. *dynamic* algorithms, but only a subset of them can be actually applied in a Web cluster. This architecture with a single Web switch that receives all incoming requests drives the choice to centralized dispatching policies. Moreover, if we consider that the main load sharing objective is to smooth out transient peak overload periods while load balancing algorithms strive to equalize the loads on all the server nodes [77, 105], it is clear that a Web switch should aim to share rather than to balance cluster workload. Absolute stability is not always necessary and sometime impossible to achieve in a highly dynamic system such as a Web cluster hosting a popular Web site.

The real alternative is therefore among static vs. dynamic algorithms, although the Web switch cannot use highly sophisticated dispatching algorithms because it has to make fast decision for hundreds or thousands of requests per second. *Static* algorithms are the fastest solution to prevent the Web switch from becoming the primary bottleneck of the Web cluster because they do not rely on the current state of the system at the time of decision making. However, these algorithms can potentially make poor assignment decisions. *Dynamic* algorithms have the potential to outperform static algorithms by using some state information to help dispatching decisions.

[1] We use the definition of *global scheduling* given in [40] and *dispatching* as synonymous.

On the other hand, dynamic algorithms require mechanisms that collect and analyze state information, thereby incurring in potentially expensive overheads. The requirements listed below summarize the constrains for dispatching algorithms that we will analyze in this Section.

1. Low computational complexity, because dispatching decisions are required to be made in real-time.

2. Full compatibility with existing Web standards and protocols.

3. All state information needed by a dispatching policy has to be actually accessible on the Web switch. In particular, the switch and servers of the Web cluster are the only entities that can collect and exchange load information. We do not consider any state information that needs active cooperation from other components that do not belong to the content provider.

4.1 A taxonomy of dispatching algorithms

We have seen that in Web clusters the only practical choice among all global scheduling policies lies in the *static* vs. *dynamic* algorithms. A third class of load sharing politics that has been widely investigated in literature is the class of *adaptive* algorithms, where the load sharing policy as well as the policy parameters can change on the basis of system and workload conditions [105]. However, to the best of our knowledge, no existing Web switch uses adaptive algorithms.

Static dispatching algorithms do not consider any state information while making assignment decisions. Instead, dynamic algorithms can take into account a variety of system state information that depends also on the OSI protocol stack layer at which the Web switch operates. Because of the importance of this factor, we prefer first to classify the dispatching algorithms among *content-blind dispatching*, if the Web switch works at TCP/IP level and *content-aware dispatching*, if the Web switch works at application level. Both content-aware and content-aware blind dispatching algorithms are based on best-effort principles. We also introduce a class of algorithms which is designed to guarantee Service Level Agreements (SLA) to specify class of Web users. This new class is called *QoS-aware dispatching*.

We then use the literature classification by distinguishing static and dynamic algorithms. It is to be noted that we assume that static algorithms are deployed only by the Web switches that operate at the TCP/IP level because the use of a sophisticated architecture such as a layer-7 switch is motivated only if its benefits are fully exploited by the dispatching algorithm. Dynamic algorithms can be further classified according to the level of system state information being used by the Web switch. We consider the following three classes.

Client state aware policies. The Web switch routes requests on the basis of some client information. Layer-4 Web switches can use only *network-level* client information such as client IP address and TCP port. On the other hand, layer-7 Web switches can examine the entire HTTP request and make decisions on the basis of more detailed information about the client.

Server state aware policies. The Web switch assigns requests on the basis of some server state information, such as current and past load condition, latency time, and availability. Furthermore, in content-aware dispatching, the switch can also take into account information on the content of the server disk caches.

Client and server state aware policies. The Web switch routes requests by combining client and server state information. Actually, most of the existing client state aware policies belong to this class, because they always use some more or less precise information about the server loads (at least server availability).

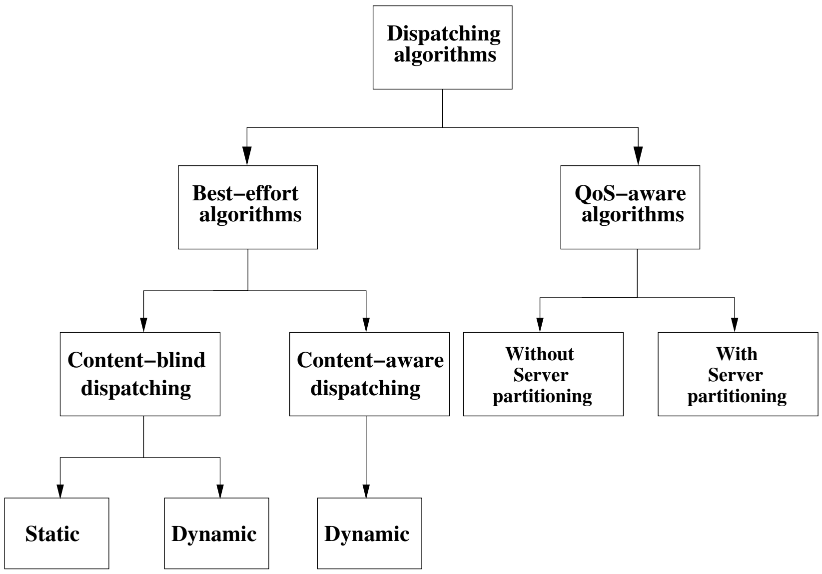


Figure 4.1: Taxonomy of dispatching politics in Web clusters.

Figure 4.1 summarizes the taxonomy for dispatching algorithms that we have examined so far. We recall that static algorithms as well as server state aware policies are meaningful only for content-blind Web switches operating at TCP/IP level.

4.2 Content-blind dispatching algorithms

In this Section we describe content-blind dispatching policies according to the taxonomy shown in Figure 4.1, and detailed in Figure 4.2 with some representative algorithms for each category at the bottom level.

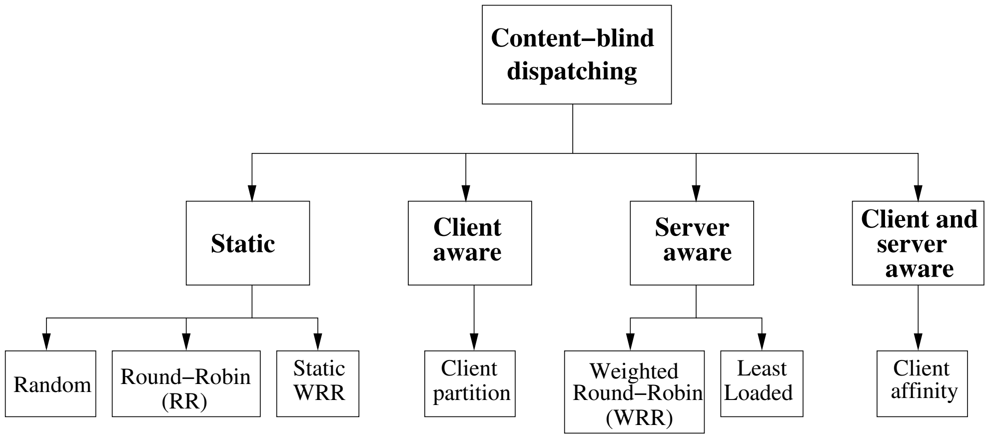


Figure 4.2: Content-blind dispatching algorithms.

4.2.1 Static algorithms

Static policies do not consider any system state information. Typical examples are **Random** and **Round-Robin** (RR) algorithms. Random distributes the incoming requests uniformly through the server nodes with equal probability of reaching any server. RR uses a circular list and a pointer to the last selected server to make dispatching decisions, that is, if S_i was the last chosen node, the new request is assigned to S_{i+1}, where $i+1 = (i+1)\, mod\, N$ and N is the number of server nodes. Therefore, RR utilizes only information on past assignment decision.

Both Random and RR policies can be easily extended to treat servers with different processing capacities by making the assignment probabilistic on the basis of the server capacity [51]. To this purpose, if C_i indicates the server capacity, the relative server capacity ξ_i ($0 \leq \xi_i \leq 1$) is defined as $\xi_i = C_i/max(C)$, where $max(C)$ is the maximum server capacity among all the server nodes. It is to be noted that the server capacity is a configuration parameter, thus a static information. For Random policy, heterogeneous capacities can be taken into account by assigning different probabilities to the servers according to their capacity. The RR policy can treat heterogeneous server nodes in the following way. A random number ϱ ($0 \leq \varrho \leq 1$) is generated, and, assuming S_i was the last chosen node, the request is assigned to S_{i+1} only if $\varrho \leq \xi_i$. Otherwise, S_{i+2} becomes the next candidate and the process recurs, that is another random number is generated and compared with the relative capacity of S_{i+2}.

Different processing capacities can be also treated by using the so-called **static Weighted RR** (for short, static WRR), which comes as a variation of the Round-Robin policy. Each server is assigned an integer weight w_i that indicates its capacity. Specifically, $w_i = C_i/min(C)$, where $min(C)$ is the minimum server capacity among all the server nodes. The dispatching sequence will be generated according to the server weights [82]. As an example, let us assume that S_1, S_2, and S_3 have the weights 3, 2, and 1, respectively. Then, a dispatching sequence can be $S_1\, S_1\, S_2\, S_1\, S_2\, S_3$.

4.2.2 Client state aware algorithms

As layer-4 Web switches are content information blind, the type of information regarding the client is limited to that contained in TCP/IP packets that is, IP source address and TCP port numbers. These coarse client information can be used to provide a simple method to statically partition the server nodes and to assign groups of clients identified through their IP address to different servers.

4.2.3 Server state aware algorithms

Client information is immediately available at the Web switch because it receives all requests for connection. On the other hand, when we consider dispatching algorithms that use some server state information we have to address issue related on server load index and gathering time, discussed in Section 4.2.5.
On the other hand, when we consider dispatching algorithms that use some server state information we have to address several issues: which *server load index*? How and when to compute it? How and when to transmit it to the Web switch? These are well known problems in networked system [54, 64]. The note in Section 4.2.5 is devoted to the analysis of some alternatives in Web clusters.

Once a server load index is selected, the Web switch can apply different dispatching algorithms. A common scheme is to have the new connection assigned to the server with the lowest load index (the so-called **Least Loaded** policy). Two example of load index commonly used are the number of active connections and the response time.

Least Connections policy, which is usually adopted in commercial products (e.g., Cisco's LocalDirector [47], F5's Networks' BIG/ip [63]) the Web switch assigns the new request to the server with the fewest active connections. A simple extension, which assigns static weights to the servers according to their capacity, allows to take into account heterogeneity in server capacity [82]. The underlying idea is that servers with greater capacity should support a larger number of active connections: this can be simply achieved by dividing the number of active connections by the server weight.

Fastest Response policy, the Web switch assigns the new connection to the server which is responding faster that is, showing the smallest latency time in the last observation interval.

The **Weighted Round-Robin** (WRR) algorithm is a variation of the static WRR, whose pseudo-code is shown in Figure 4.3.

```
while (true) {
    fetch next request r;
    chosen_server = 0;
    min_load = load(chosen_server);
    for (i=1; i<num_servers; i++) {
        cmp_server = i;
        if (min_load > load(cmp_server)) {
        chosen_server = cmp_server;
        min_load = load(cmp_server);
        }
    }
    load(chosen_server)++;
    send r to chosen_server;
}
```

Figure 4.3: Pseudocode of the WRR algorithm

WRR associates each server with a dynamically evaluated weight that is proportional to the server load state [73]. Periodically, every T_{get} seconds, the Web switch gathers load index information from the servers and computes the weights, that are dynamically incremented for each new connection assignment. As mentioned in Section 4.2.3, the server load information *load(cmp_ server)* (see the pseudocode in figure 4.3) may be the number of active connections (WRR_num) or the request service time (WRR_time).

4.2.4 Client and server state aware algorithms

Client information available at a layer-4 Web switch (e.g., the source IP address and the service port number within the TCP header) is usually combined with some server state information to provide the so called *client affinity* [73, 82]. Instead of assigning each new connection to a server only on the basis of the server state regardless of any past assignment, consecutive connections from the same client can be assigned to the same server for either performance or functional reasons. As an example of performance reasons, consecutive SSL connections from the same client are assigned to the same server during the life span of the SSL key, so to avoid time- and resource-consuming operations for SSL key negotiation and generation. An example of functional requirement for client affinity is FTP, as this protocol uses two connections for the same client-server interaction. In policies based on client affinity, the client and server information have a different weight: the client information, when available, usually overrides server information for assignment decisions.

4.2.5 A note on server state information

Various issues need to be addressed when we consider dispatching policies based on some server state information:

(i) the choice for one or more *server load index(es)*;

(ii) the way to compute the load state information and the frequency of the samples;

(iii) due to the cluster architecture, some indexes may not be immediately available at the Web switch, so we have to decide how to transmit them and how frequently.

Any of these choices has a deep impact on final performance of the dispatching algorithms, to the extent that the same policy can behave much better or much worse of other algorithms depending on the quality of the chosen load indexes.

The three main factors that affect the latency time of a Web request are loads on CPU, disk, and network resources of the Web server nodes. Typical server state information includes the CPU utilization evaluated over a short interval, the instantaneous CPU queue length periodically observed, the amount of available memory, the disk or I/O storage utilization, the instantaneous number of active connections, the number of active processes, and the object latency time that is, the completion time of an object request in the Web cluster.

Load indexes can be further classified into three classes according to the way they are evaluated: *input indexes*, *server indexes*, and *forward indexes*.

Input indexes are computed by the Web switch and do not require any server cooperation.

Server indexes are evaluated by each server and transmitted to the Web switch.

Forward indexes are information got directly by the Web switch that emulates HTTP requests to the Web servers.

The Web cluster architecture determines the feasibility and convenience of using some load indexes instead of others. For example, input indexes and server indexes can be used in one-way and two-way architectures while forward indexes are meaningful in one-way architectures only, because in two-way architectures the Web switch can keep track of each connection without the need of generating additional traffic in the Web cluster.

Both server and forward indexes typically take longer than input indexes to acquire. However, input index information is limited to the number of active connections that provides a rough estimate of the state of the Web servers as seen by the binding table of the Web switch.

On the other hand, server indexes can provide detailed information such as CPU and disk utilization, object latency time, number of active connections, number of processed packets. This last information can be the most useful load index when the number of transferred packets varies greatly from connection to connection, for example, when large files can be transmitted or when HTTP/1.1 persistent connections are used. In one-way architecture, server indexes have to be computed by a process monitor running on each server and periodically transmitted to (or get by) the Web switch. In two-way cluster architectures the same mechanism can be used, otherwise a subset of server indexes (e.g., number of active connections, transmitted packets, object latency time) can be inferred by the Web switch that has a full control on the data flow. Other server indexes, such as CPU and disk utilization, always need a process monitor on the servers and a communication mechanism from the servers to the Web switch.

It is to be noted that the combination of a set of load indexes into a single index that reflects the server load is an interesting research issue that has not yet been investigated in the context of Web clusters.

In addition to the choice of the server load index, all server state aware policies face the problem of updating the load information. The intervals between updates of the load indexes need to be evaluated carefully to make sure that the system remains stable. If the interval is too long, performance may be poor because the system is responding to old information about the server loads. On the other hand, too short intervals can result in system overreaction and instability. A simple strategy for interpreting stale load information that can apply to layer-4 Web switches has been proposed in [54].

4.3 Content-aware dispatching algorithms

The complexity of layer-7 Web switches that can examine the HTTP request motivates the use of more sophisticated content-aware distribution politics. In this Section we detail the taxonomy for content-aware dispatching shown in Figure 4.1 with an additional level that considers the main goal of the dispatching politics. Figure 4.4 summarizes the taxonomy of the content-aware dispatching policies and shows at the bottom level the proposed algorithms that use information about the requested URL for different purposes, such as

- to augment reference locality in the server caches so to reduce disk accesses (*cache affinity*);
- to use specialized server nodes to provide different Web services (*specialized servers*), such as streaming content, dynamic content, and to partition the Web content among the servers, for increasing secondary storage scalability;
- to augment load sharing among the server nodes (*load sharing*).

Furthermore, additional information regarding the HTTP request, such as cookies and SSL identifiers, can be also used to exploit *client affinity* algorithms. Indeed, the

SSL protocol involves a computationally expensive handshake procedure (certificates exchange, encryption and compression negotiation, session ID setup), while subsequent SSL sessions can skip the handshake (by using again the same session ID) for a limited period of time. However, it is to be noted that support for stateful services can be also provided by Web switches that operate at TCP/IP level (although with a lower degree accuracy as explained in Section 4.3). Indeed, stateful services can be identified through the service port (e.g., 443 for SSL) and a connection reuse timeout can be set [73].

Finally, when HTTP/1.1 persistent connections are used, a layer-7 Web switch can assign requests traveling on the same TCP connection to different servers, thus achieving a granularity control down to individual HTTP requests. On the other hand, a layer-4 switch must assign the entire TCP connection to the same server. It implies that multiple HTTP requests on the single persistent connection reach the same server, that is the control granularity on which the assignment is activated is at the level of the entire TCP connection. With HTTP/1.0, there is no difference for the granularity control between layer-4 and layer-7 routing because a one-to-one correspondence exists between an HTTP request and a TCP connection.

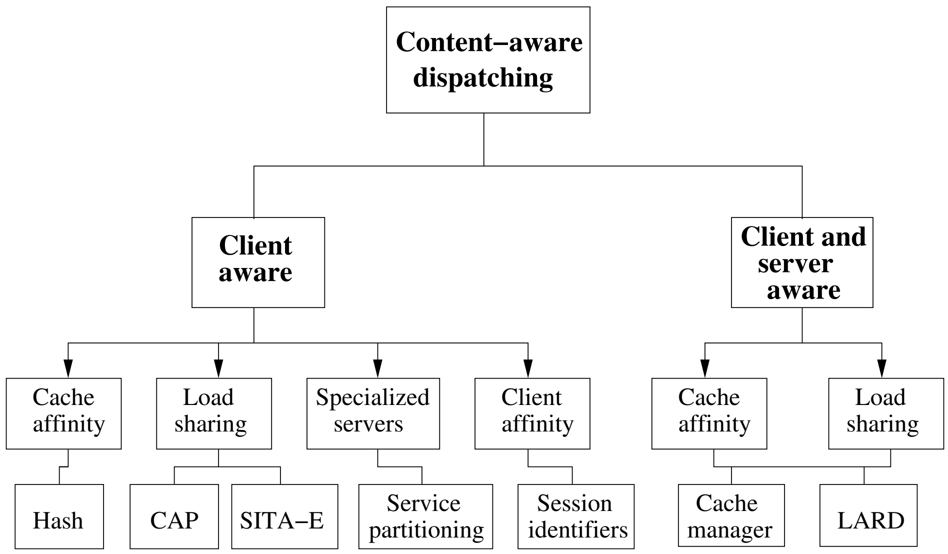


Figure 4.4: Content-aware dispatching algorithms.

4.3.1 Client state aware algorithms

Let us first consider the left part of the taxonomy in Figure 4.4. In cache affinity policies, the file space is typically partitioned among the server nodes. A hash function can be used to perform a static partitioning of the files. The dispatching policy running on the Web switch (namely, **Hash** algorithm) uses the same function. This scheme exploits at maximum the locality of references in the server nodes and achieves the best cache hit rate. However, it can be applied to Web sites providing static content only. Moreover, it ignores load sharing completely, as it is difficult to partition the file space in such a way that the requests are balanced out. Indeed, if a small set of files accounts for a large fraction of requests (a well-know characteristic of Web workload, e.g., [13, 53]), the server nodes serving those critical files will be more loaded than others.

For Web sites providing heterogeneous Web services, the requested URL can be used to statically partition the servers according to the service type they handle. The goal is to employ specialized servers for certain type of requests, such as dynamic content, multimedia files, streaming video [113]. We refer to this policy as to **Service Partitioning**. Most commercial content-aware switches deploy this type of approach (e.g., F5's BIG-IP [63], Resonate's Central Dispatch [102]).

The third main goal of the content-aware dispatching algorithms is to improve load sharing among the server nodes. These strategies do not require a static partition of the file space and the Web services. Two policies belong to this class: **Size Interval Task Assignment with Equal load** (SITA-E) [71] and **Client-Aware Policy** (CAP) [39].
SITA-E is more oriented to Web sites providing static information; CAP, that is the new algorithm proposed in this thesis, is more oriented to sites providing Web services with different computational impact on system resources.

The **SITA-E** policy partitions dynamically Web content among the servers according to the file size distribution. It defines the size range associated with each server in such a way that the total load directed to each server is the same [71]. The Web switch determines the size of the requested file and selects the target server on the basis of this information. The goal is to assign light tasks to the light-loaded nodes thus separating light from heavy tasks. The SITA-E policy founds on the basic assumption that the service time of a request is proportional to its size; this assumption is, however, valid for static content only.

Most load sharing problems occur when the Web site provides heterogeneous services that make an intensive use of different Web server resources. Previously described policies do not consider this issue as they either focus on the provision of static content or address multiple services through a static server partitioning. To improve load sharing in Web clusters that provide multiple services, the class of **CAP** policies takes into consideration the requested service [39]. Web requests are classified on the basis of their impact on main Web server resources, such as static, lightly dynamic, intensive dynamic (e.g., disk bound services, CPU bound services, disk and CPU bound services). Although the Web switch cannot estimate the service time of a client request accurately, it can distinguish the class of the request from the URL and estimate its impact on main Web server resources. The Web switch manages a circular list of server assignments for each class of Web services.

The CAP class will be detailed in Chapter 9 where we present performance studies and comparisons with other content-aware dispatching algorithms.

The last type of client information that the Web switch can use to select a server is the so-called **session identifier**, like cookies and SSL identifiers. Based on this information, the Web switch can assign all Web transactions from the same client to the same server. Session identifiers provide a powerful means to maintain client affinity at the individual client granularity. In particular, they avoid the limitations of the client IP address identification at a layer-4 Web switch, when Web proxies in the client-server path can squeeze a large number of users into a small number of different IP addresses.

Pure client aware poilicies do not require a hard tuning of parameters, which is typical of most dynamic policies, because the service classes are decided in advance and the dispatching choice is determined statically once the requested URL has been classified.

4.3.2 Client and server state aware algorithms

Dispatching algorithms implemented at application level can also use a combination of client and server state information. In this Section, we describe two policies that have been specifically designed to consider both client and server information. Other client aware policies (e.g., CAP) can be easily integrated with some server state information. In the taxonomy in Figure 4.4, the proposed policies belong to two classes because they use client information for cache affinity purposes and server information for load sharing goals.

The **Locality-Aware Request Distribution** (LARD) policy is a content-aware request distribution that considers both locality and load balancing [15, 96]. The basic principle of LARD is to direct all requests for the same Web object to the same server node until its utilization is below a given threshold. By so doing, the requested object is more likely to be found into the disk cache of the server node. Some check on the server utilization is useful to avoid overloading servers and, indirectly, to improve load sharing. When a server utilization reaches a given watermark, the dispatcher assigns the request to a lowly loaded node, if it exists, or to the least loaded server. A scheme similar to the LARD policy has been implemented also in the HACC cluster [115].

While in LARD policy the Web switch maintains the mapping from a given file to a set of nodes that serve that file, the **Cache manager** dispatching policy relies on a cache manager that is aware of cache content of all Web servers [29]. Each server provides periodically this information to the cache manager. If the requested object is not cached in any server, the Web switch selects the least loaded server. Otherwise, it selects the lightest loaded server having the object cached, provided that its load is within a threshold over the least loaded server [29].

4.4 QoS-aware dispatching algorithms

4.4.1 Motivations

The second generation of Web sites provides more complex services than those related to Web publishing sites. Many users already rely on the Web for up-to-date personal and business information and transactions. so the need for differentiated users and services is becoming stronger. This differentiation is desired to accommodate heterogeneous application requirements and user expectations, and to permit differentiated pricing for content hosting or service providing. For example, economic transactions are more important than simple browsing, internal or premium service users may expect a better treatment, as well as the owner of a co-located Web site that paid a higher fee.

In this Chapter, we introduce the concept of “Quality of Web Services” (*QoWS*) that is inspired by the well known QoS principles for networks, but it is focused on the server side of the Web system. In particular, we investigate how QoWS mechanisms and principles can be applied to a Web site that is hosted on a system platform consisting of locally distributed Web server nodes, namely Web-server cluster or *Web cluster* in short.

Although in this thesis we look at the server side only, we acknowledge that it would be better to look at the whole system, including the network and the client application. Indeed, a network congestion, insufficient performance of the browser, a badly connected user machine would provide a bad user experience despite the Web cluster performance. There is no doubt that the last challenge for the research in this field is to combine network QoS and server QoWS. However, in this thesis we address QoWS issues for Web clusters to focus on the server side only, and to assume that network and client problems are solved elsewhere.

QoWS-enabled systems for Web services have been recently proposed for single and multiple servers platforms. For a Web site hosted on a single server node, the main actions focus on scheduling algorithms and resource management of server components, both at the application (Web server) and operating system level. Most results for QoWS-enabled clusters consider Web content hosting that is, multiple Web sites co-located on the same platform. On the other hand, we want to investigate mechanisms and policies for a cluster that hosts a single Web site. Very few results exist on this last topic [43, 76, 117] and all of them aim to compare the proposed QoWS policy against a not QoWS-enabled Web cluster. Our point of view is more taxonomic as we first extend basic QoS principles to QoWS and then consider most policies that combine all or part of QoWS principles. Actually, in this thesis we do not present all possible combinations but focus on some representative classes of policies that in preliminary experiments gave best results.

At the base of QoS (and QoWS as well) there are the concepts of *service* and *Service Level Agreement* (SLA). A *service* defines a set of characteristics significant for the Web content provision, specified in quantitative or statistical terms. In particular, in this thesis we consider two classes of users (i.e., *High class* and *Low class*) and two main classes of Web requests (i.e., *static requests* and *dynamic requests*). An SLA is a specified performance contract agreed between the user and the service provider. Depending on its strictness, it may be either a *guaranteed service* if the resources are reserved to never violate the agreement (contract), a *predictive service* if rare violation are allowed and the service is expressed in terms of percentile, or a *best effort service* if the service is provided with no contractual target. Web servers, together with the network, are parts of a soft real-time system that has to provide information and services to the browsers. The issuer of the request takes into account the time and the request value decreases as time goes on, however there are no catastrophic consequences if a deterministic deadline is not met as in hard real-time systems. For this reason, we consider highly acceptable to express the SLA for QoWS in terms of predictive service, measured as 95-percentile on some performance values [79]. Other authors consider for SLA quite different parameters such as user satisfaction and economic returns [87]. Our goal is to find a user-oriented evaluation but independent of specific human parameters. For this reason, we will consider the 95-percentile of *latency time of a client request* at the Web cluster as the main metric for SLA. Web cluster throughput and percentage of request dropping will be used as additional metrics for comparing different QoWS solutions.

The necessity for QoS was born in the computer network area, where this topic is highly investigated. In [80], there is a clear formulation of the basic principles necessary to provide QoS guarantees to network applications. Let us recall these principles:

- classification of services;
- performance isolation;
- high resource utilization;
- request admission, that includes a first step where a request "declares" its needs (namely, *declaration*), a verification of the availability of resources for satisfying these requirements, and a decision about access or blocking (namely, *access control*).

In computer networks, integrated and differentiated services are well known mechanisms to guarantee QoS through these underlying principles [26, 25]. As our research guideline to enable QoWS in Web clusters, we consider useful to follow three main steps.

1. It is necessary to find out the feasible mechanisms that can be employed to achieve QoWS in a Web cluster.
   Our analysis leads to select switch operations at layer-4, switch operations at layer-7, request dispatching among servers, request dropping, and resource scheduling.

2. It is also necessary to find out the components of a Web cluster that can implement the principle and relative mechanisms.
   The main candidates for implementing QoWS mechanisms are the Web switch

(working at layer-4 or layer-7) for centralized solutions and each Web or back-end node for distributed solutions.

3. The next steps are the extension of the QoS principles to QoWS principles in Web clusters and the individuation of suitable policies. The analysis of QoWS principles is done below, while the study of policies in Sections 4.4.3 and 4.4.4.

4.4.2 A methodology for QoS in Web cluster

In extending QoS to QoWS principles in a cluster, it is important to precise the definition of "resource". In a single Web server architecture, a "resource" is typically one of the server components, such as CPU, disk, and network interface. This definition of resource is still applicable to Web clusters, however these architectures can use even a more coarse-grained definition, such as considering an entire server as a "resource". While the fine-grained interpretation is typically used for addressing QoWS in single server systems [19, 28], in this Chapter, if not otherwise specified, a resource is a (Web or back-end) server.

Classification. The first step to differentiate the service is to classify the incoming requests in order to bind each one with the appropriate class of service. Depending on the component of the Web cluster executing the classification (i.e., layer-4 Web switch, layer-7 Web switch, server), new requests can be distinguished depending on the location, the requested service, bounded information like cookies or application specific parameters. The classification mechanism of a layer-7 Web switch can be based on all information at the application level. This allows a switch to achieve the most detailed classification analysis including client classification, user identification, and request classification. (From this point of view, a Web server is analogous to a layer-7 switch.) On the other hand, a layer-4 Web switch working at TCP/IP level can provide a limited client classification only, that is based on information contained in TCP/IP packets. Since a precise classification is at the basis of any QoWS principle, the above analysis should be sufficient to motivate the need to use a layer-7 Web switch in order to achieve the best request/user differentiation.

Performance isolation. To force different classes of users and requests to share the resources, it is necessary to provide a degree of isolation among classes of service. Performance isolation can be achieved through queuing scheduling

policies [57] and/or resource partitioning mechanisms. A scheduling priority mechanism applied to all resources could allow over aggressive classes of services to monopolize the entire system. In this paper, we limit the priority scheduling to some server components, basically the CPU.
The partitioning mechanism can be applied at different levels of granularity. While for single Web server architectures, resource partitioning is typically done at the level of server components, we think that in a Web cluster it is sufficient as a first step to consider an entire server as a resource. This makes also easier the applicability of QoWS principles to commodity off-the-shelf software. As a consequence, the most immediate performance isolation policy in a Web cluster is to have a partition of the servers depending on the classification of services. For example, with two classes of users and two classes of requests it seems convenient to use a subset of servers for each class of users, and Web servers and back-end nodes for static and dynamic requests, respectively.

High resource utilization. To use the system resources as efficiently as possible, we have to make unused resources available to other classes of services. A way to satisfy this principle is to have a dynamic server partition. Dynamic partitioning outperforms the static one since it obtains better resource utilization still maintaining good service class isolation [117]. By referring to the previous example, the server partition between two sets would be dynamically determined so that the cardinality of each set could vary, for example depending on the type of offered load during a certain period. It is immediate to get that any static resource allocation could not guarantee high resource utilization because the percentage of services from different classes cannot be considered static during the entire day. Moreover, because of the high variability of Web traffic, to guarantee SLAs for the classes of privileged users would require a system dimensioning on peak usage for each combination of offered load: this solution would be unrealistic and expensive.

Request admission. An admission process is needed to avoid Web cluster overloading and also to allow it to meet the predictive service targets. This process consists of two steps: declaration and access control.

- A declaration process with the purpose of estimating the resources needed by a request can be realistically carried out only by a layer-7 Web switch or a Web server, because they are content information aware. For example, it is important to differentiate at least static from dynamic requests, because dynamic requests can easily require service times of two or more orders of magnitude higher than static requests. On the other hand, a layer-4 Web switch cannot implement a real declaration process because all of its information is at the TCP level. (The port number can be useful if the cluster provides multiple network services, such as FTP, SMTP, SSL. However, this is not the case considered in this thesis.)

- After the classification of a service, if there was a contract for it, the access control mechanism should check whether there are sufficient resources to satisfy SLA and decide about allowing or dropping connection. Admission control is essential to not overload the servers, as overloaded servers experience a significant loss of throughput [45]. Hence, the most important information is to know the state of the servers (e.g., utilization of single server components, number of active connections). As we are considering one-way Web cluster architectures, the Web switch is not aware about the system state unless each server periodically sends this information to the switch or the switch gathers it from the servers. This mechanism for exchanging server state information does not put additional overhead on the system, because it already exists for dynamically dispatching purposes (see WRR policy in Section 8). Layer-4 and layer-7 Web switches use connection dropping as the access control mechanism. However, due to the different quality of the request/user classification, this dropping mechanism can be applied in quite different ways. We will present some alternative policies in the next section.

  A server has the same content information as a layer-7 switch for the declaration phase, while it is relatively penalized for the access control phase. Indeed, unless introducing cumbersome all-to-all communications among servers, each server has a precise but local view on the system state. For these reasons and because of a certain degree of inequality of load distribution among servers, any admission policy implemented at the Web server level would not be as fair and efficient as centralized policies at the dispatcher level. On the other hand, for some critical cases (e.g., the decision about the request rejection of a high class user), it is important to have the most precise information about the server load. In these instances, a server admission control could work better than a centralized policy with possible not updated state information.

From the above qualitative analysis, we have that the only way to guarantee all basic QoS principles for QoWS in a Web cluster is an approach that uses a layer-7 Web switch for classification and admission control, and a dynamic server partition for performance isolation and high utilization.

Since each Web server in the cluster can be enhanced to support QoWS by acting on the server resources scheduling, we also explore the combination of Web switch policies with priority CPU scheduling at each Web server and back-end node.

In the following sections we propose different QoWS policies that satisfy all or part of QoS principles.

4.4.3 QoS-aware policies without server partition

If we do not provide server partition, each incoming request can be assigned to any server, independently of the service class it belongs to. To satisfy the SLA of the most demanding classes, the basic idea of the policies without server partition is to deny service to the requests of low class users when the cluster load exceeds a given threshold, and reject the requests of high class users only in the case of extremely critical load conditions.

We classify the policies that do not partition server nodes on the basis of the component that implements admission control (that is, centralized at the Web switch or distributed at the servers) and whether a distributed performance isolation mechanism is implemented or not. Indeed, the admission control could be integrated with a CPU scheduling mechanism that gives service priority to the requests of the high class users. The combination of the alternative solutions originate four main classes of policies that satisfy two (classification, request admission) or three (performance isolation too) QoWS principles. However, they do not introduced any explicit mechanism to guarantee high resource utilization, besides some load sharing provided by the WRR dispatching policy.

SwitchAdmission. In this class of policies, the admission control mechanism is implemented by the Web switch. As a representative member of this class, we consider the **SwitchAdm** algorithm that defers the service of the requests belonging to the lower class of users. The rejection mechanism is triggered by the

Web switch when the current sum of the server load (*SumLoad*) in the cluster exceeds a given threshold (Thr). This threshold is set to $\alpha \cdot N \cdot MaxConn$, where $0 < \alpha \leq 1$, N is the number of Web servers, and $MaxConn$ represents the maximum number of active connections that each server can sustain without performance degradation. Every T_{get} seconds, the Web switch gathers the number of active connections from each server, calculates *SumLoad*, and determines if the requests have to be dropped or accepted. The rejection phase starts when the sum of the servers load exceeds the threshold Thr and ends when the sum returns under the same threshold value. Therefore, this policy is equivalent to a QoWS-blind dispatching policy enhanced with an admission control mechanism to prevent performance degradation [45]. In the experiments, we set $\alpha = 0.9$ as we found in preliminary experiments that such a value does not affect significantly the latency time while it allows to reduce the number of dropped requests. The same threshold is used for switching on and off the dropping phase, because experiments with different threshold values did not improve the performance of the SwitchAdm policy.

ServerAdmission. The rejection mechanism could be independently triggered by each server when its load exceeds the maximum number of active connections. Letting each server control the admission of the requests did not give acceptable results, so we will not consider any policy that provides a distributed access control solution. It seems that a global view on the cluster load is quite preferable to decide about admission/rejection instead of taking decisions on a local view of the server state. This conclusion is valid even if the server works on the most updated load information, while the Web switch information, although frequently updated, cannot be as precise as that available at each server.

SwitchAdmission-ServerPriority. This class combines classification and request admission control at the Web switch (centralized solution) with a performance isolation principle carried out by the servers (distributed solution). In this thesis, we consider the **SwitchPri** algorithm that uses the SwitchAdm policy at the Web switch and the priority scheduling mechanism for the requests of high class of users at the CPU of the Web servers and back-end nodes.

ServerAdmission-ServerPriority. The last alternative is to combine a distributed admission control mechanism carried out by each Web server with a distributed performance isolation policy such as priority scheduling at the servers. Because of the poor results of these solutions, we do not consider any policy of this class in the thesis.

4.4.4 QoS-aware policies with server partition

Performance isolation is one of the main principles that we borrowed from the QoS theory. A simple way to implement it in a Web cluster is through a centralized Web switch mechanism that partitions the servers in as many classes as the defined services. This solution intervenes on the dispatching algorithm and does not require modifications either at the application or operating system level of the servers. In the hypothesis of two classes of users, we partition the servers into two sets, denoted as *High Set* (HS) and *Low Set* (LS). Incoming user requests classified as high are assigned to servers in the first set, while servers in LS serve user requests belonging to the low class. The corresponding admission control policy can reject requests belonging to the low class only, unless the Web cluster is really overloaded.

An alternative approach to the centralized solution provided by server partition is to implement performance isolation through distributed mechanisms. Resource scheduling algorithms and request redirection are examples of solutions where each server can implement performance isolation. In this thesis we consider some scheduling solutions, however it is important to note that these mechanisms require to modify the server application and/or the operating system. Request redirection solutions are feasible, but we discard them because they add a significant overhead to each request. The basic idea is that any server can receive any request and, if necessary because of the specific QoWS strategy, it can redirect it to another server through HTTP or TCP tunneling mechanisms [8, 18, 36]. Another motivation for excluding request redirection solutions is that they do not guarantee a perfect isolation of High Set servers from low priority requests. This is even more evident if we use a tunneling mechanism because the first contacted server has always some role in the packet triangulation mechanism [99].

Performance isolation can be combined or not with the high resource utilization principle. This alternative originates two sets of policies. We propose a representative policy for each set, even if it is important to observe that many alternative algorithms exist. Some of them have been proposed in literature [117], others have not been explored yet, so that we can state that there is still room for research in this field. Both policies we are considering use *SwitchAdm* as the request admission at the Web switch.

StaticPartitioning. In this class of strategies, the cardinality of the High and Low set of servers is set to an initial value that remains constant until the Web administrator changes it. We consider the **StaticPart** algorithm that works as following. Let assume that there are N servers in the cluster. At a given time t, it results $HS(t) = \{1, \ldots, us(t)\}$ and $LS(t) = \{us(t)+1, \ldots, N\}$, where $HS(t) \cap LS(t) = \emptyset$, and $us(t) \in \{1, \ldots, N\}$. The upper limit of HS, that is $us(t)$, is initially set to $us(0)$. The problem is to determine the best value for $us(0)$ that allows the Web cluster to satisfy most SLA requirements. The trivial solution is to set $us(0) = \lceil \rho N \rceil$, where ρ represents the percentage of connection requests from users belonging to the high class. This means that the servers in the $HS(t)$ partition handle $\lceil \rho N \rceil \cdot MaxConn(T_N)$ connections per second, where T_N is an admissible latency time for a request, and $MaxConn(T_N)$ is the maximum number of connections that the server can sustain with a latency time of T_N seconds. However, if we have to satisfy a SLA of T_H seconds, the value of the previously evaluated $us(0)$ must be incremented of x units, where x is the appropriate number of servers such that

$$\lceil \rho N \rceil \cdot MaxConn(T_N) = (\lceil \rho N \rceil + x) \cdot MaxConn(T_H) \tag{4.1}$$

From Equation 4.1, it results that

$$us(0) = \lceil \rho N \rceil \cdot \frac{MaxConn(T_N)}{MaxConn(T_H)}. \tag{4.2}$$

The value of $us(0)$ remains constant until the Web administrator realizes that the configuration of the Web cluster is highly inadequate.

DynamicPartitioning. This is a large class of policies for which the initial value for $us(0)$ is determined as in the StaticPart policy, but we introduce now a mechanism to adapt the size of each partition to the actual workload composition and server load state. In this thesis, we consider a basic policy, namely **DynamicPart**, and we propose some variants in Section 10.3.

To describe DynamicPart, let us suppose that at time t the size of the HS partition is $us(t-1)$ and the sum of the servers load in $HS(t-1)$ is $SumLoad_{HS}(t)$. We will add the least loaded server of $LS(t)$ to $HS(t)$ that is, $us(t) = us(t-1) + 1$, if $SumLoad_{HS}(t) > \alpha \cdot us(t-1) \cdot MaxConn(T_H)$. From that point, the moved server will receive only new requests from the high class users, while it will continue to serve requests of already accepted connections belonging to the low class. (Stronger actions can be used, such as dropping all pending low class requests, but we did not used them because highly unfair solutions.) The moved server will return to $LS(t)$ that is, $us(t) = us(t-1) - 1$, when $SumLoad_{HS}(t) < \alpha \cdot (us(t-1) - 1) \cdot MaxConn(T_H)$. In the basic version of the DynamicPart policy, only requests from the low class of users can be refused when $SumLoad_{LS}(t) > \gamma \cdot (N - us(t)) \cdot MaxConn(T_R)$. The parameter γ permits to regulate the ratio between the number of dropped requests (low class requests) and the load state of the server in the LS partition.

Table 4.1 summarizes in which way the above described policies satisfy the QoWS principles analyzed in Section **??** that is, service classification, performance isolation, high resource utilization, and request admission. All the considered policies satisfy at least the classification and request admission principles. Performance isolation is guaranteed either through some priority-based scheduling of server resources (e.g., SwitchPri) or through a server partition mechanism (e.g., StaticPart and DynamicPart). From this table, we see that only the DynamicPart policy introduces explicit mechanisms to satisfy all four QoWS principles, including high resource utilization through dynamic server partitioning. SwitchPri satisfies all QoWS principles even if high resource utilization is achieved only through indirect load sharing results due to WRR dispatching. On the other hand, both SwitchAdm and StaticPart do not guarantee performance isolation and high resource utilization. These observations lead us to expect that DynamicPart and SwitchPri can achieve best results for QoWS. Experiments in Section 10.3 will confirm this hypothesis.

4.4.5 Multiple-nodes Web servers

A Web cluster can be used as a platform for hosting multiple Web sites or a single site with many accesses. QoWS supports have been proposed for both systems. However we analyze solutions for Web clusters that support a single site because this is the focus of this thesis.

One of the first study on this subject is by Chen et al. [43] that have evaluated the impact of the request admission and dispatching mechanisms. Their simulation results clearly show that when the system is highly utilized, differentiated services provide better performance than those achieved by traditional Web clusters.

Kanodia et al. have recently proposed a QoWS policy that uses both admission

	Service Classification	**Performance Isolation**	**High Resource Utilization**	**Request Admission**	
				Declaration	**Admission Control**
SwitchAdm	L7-switch		WRR	HTTP info	switch dropping
Server(Adm)	server		WRR	HTTP info	server dropping
SwitchPri	L7-switch	scheduling	WRR	HTTP info	switch dropping
Server(Adm-Pri)	server	scheduling	WRR	HTTP info	server dropping
StaticPart	L7-switch	partitioning		HTTP info	switch dropping
DynamicPart	L7-switch	partitioning	dynamic partition	HTTP info	switch dropping

Table 4.1: Some policies for QoWS in a Web cluster. (Policies experimented in Section 10.3 are written in bold, classes of policies for which we do not consider any example are in italic.)

control and performance isolation mechanisms to guarantee that different classes of service have latencies within pre-specified targets. Unlike our study that focus on static and dynamic requests, they address Web sites with static content only [76].

A dynamic resource partitioning algorithm for static and dynamic Web requests is proposed in [117]. QoWS principles are addressed as an optimization problem. The consequence is that the proposed policy is rather complex and requires a large set of system parameter values that are not always easy to get. Nevertheless, their experiments demonstrate that dynamic server partitioning always outperforms static server partitioning. We confirm their conclusion for what concerns static vs. dynamic partitioning. However, we analyze and compare several mechanisms and policies through a more global view on the problem of QoWS for Web clusters.

An interesting work from Aron et al. has extended the resource principal abstraction to multi-node Web servers [14]. The authors have applied to Web clusters the "cluster reserves" concept they used for single Web servers [19]. These authors too address QoWS issues as an optimization problem. Moreover, as the amount of resources to be assigned to each service class is assumed to be known a priori, their solutions can realistically be applied to problems of Web hosting and co-location in clusters. Other QoWS supports for Web hosting have been investigated mainly to assign differentiated priorities to the requests according to which site is accessed. Many results show that simple strategies such as controlling the number of processes can improve the response time of high-priority requests while not penalizing the system throughput [4]. To enforce SLA constraints, Pandey et al. [98] examine selective allocation of server resources by assigning different priorities to the page requests.

Several companies now commercialize as their most recent products content-aware Web switches which can be used for service differentiation in Web clusters (e.g., Nortel Networks' Alteon WebOS [93], F5's BigIP [63], Foundry's ServerIron [65], Resonate's Central Dispatch [102]). Only very simple mechanisms are provided by these switches. Differentiation is typically based either on the request source or the requested service [63, 102]. Even worse, for all of them service differentiation means to statically partition server nodes and to assign different classes of requests to different server subset. Our paper and other results in literature [117] demonstrate that static partitioning cannot adapt to fluctuating arrival rates and servers load conditions. Moreover, it may lead to waste of resources when some partitions are not fully utilized while others might be overloaded.

We carried out some experiments that showed that only the class of policies that satisfies all QoWS principles (DynamicPart belongs to this class) was able to meet the service level agreement for different load and system conditions. Indeed, even the violation of a single QoWS principle prevents the Web cluster to satisfy the service level agreement in a highly variable scenario as that of a Web site system.

Additional comments on simulation and prototype results could be found in [31, 32] and [7] respectively.

Chapter 5

Simulation model

In this Chapter we propose a simulation model (based on queuing networks) of a cluster-based Web systems, which is designed to evaluate the performance of dispatching algorithms. The model will be solved throughout a discrete-event simulator. Indeed, a discrete event-driven simulator, **WebSim** discussed in section 5.2. This simulator has been designed and implemented with the purpose of giving a performance evaluation tool, dedicated to dispatching algorithms for layer-4/7 Web switches, based on some of the routing mechanisms presented in Chapter 3.

Our simulation approach to the resolution of the queue network is justified by the complexity of the workload model, discussed in Chapter 7, that may not be resolved analytically.

5.1 System model

A Web cluster is a system composed of multiple Web servers nodes, back-end nodes and a dedicated machine that acts as a Web switch. All these system components are tipically interconnected through a large bandwidth network, at least 100Mb/sec. Being the focus of this thesis on cluster-based Web system performance, the model details on Web cluster components and does not consider some external system components like the DNS servers and routers and other network components disseminated through the Internet. However, Web client behavior is modeled and network delays due to the Web switch NIC capacity are considered.

The model is shown in Figure 5.1, it refers to a one-way architecture system, as the data flow between the cluster nodes and the Web clients shows. We choose to present only this specific model, that is a subset of the model implemented in the simulator, because in Chapter 8 and 9 we will evaluate only the performance of one-way architectures. However, the WebSim tool is able to model also two-way architecture.

Now we describe in detail the main system component: Web switch, Web server, back-end server and Web clients.

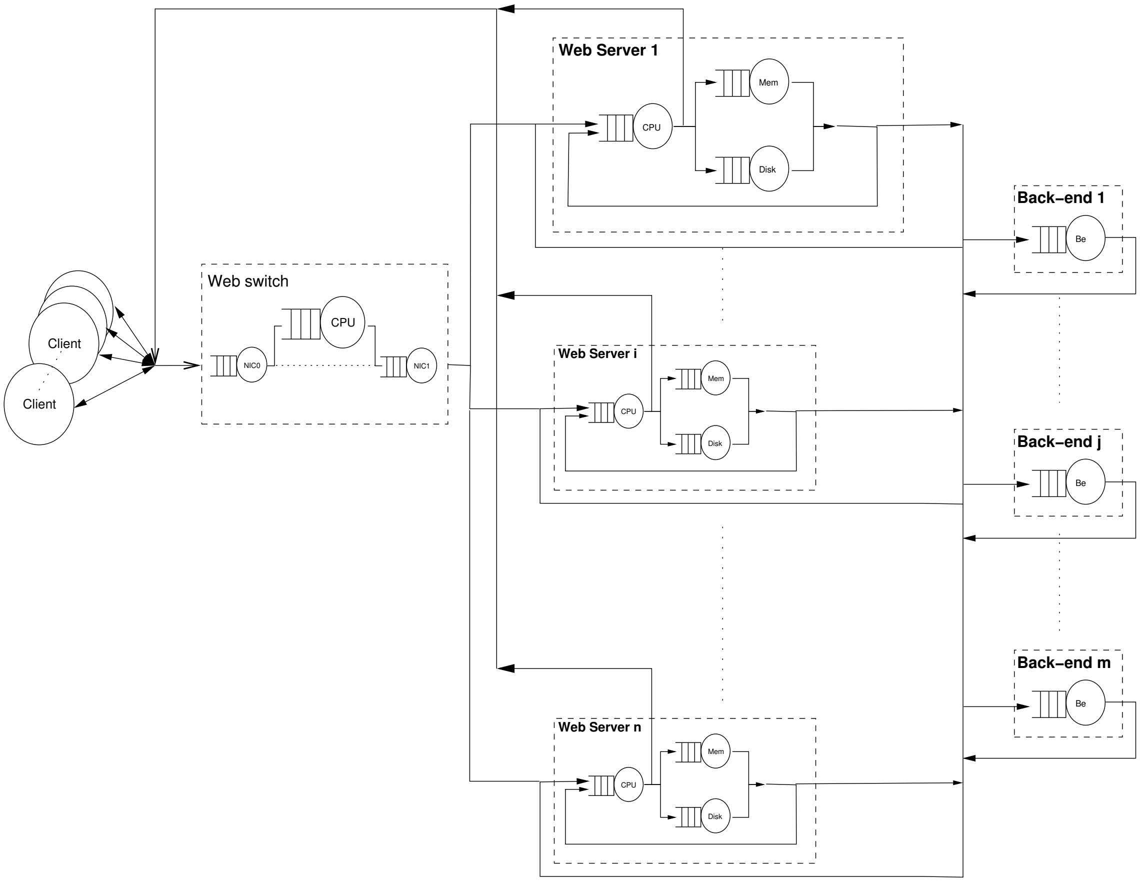


Figure 5.1: Queue network model of a cluster-based Web system.

5.1.1 Web switch model

A Web switch could be realized with a general purpose PC or with a dedicated hardware device, however in both case the components relevant to our performance study are CPU(s) and NICs. Indeed, the Web switch model proposed consists of a queue system of three service centers that, as shown in Figure 5.1, are connected to model a one-way Web switch:

NIC0 service center models the network interface connecting the Web system to the

Internet, and receiving the Web client requests;

NIC1 service center models the network interface used by the Web switch to route incoming packets over the Web cluster interconnection network;

CPU service center models the CPU used to run the operating system, the routing mechanism and the dispatching algorithms (We don not consider the case of a multi-processors device.)

A two-way Web switch, could be modeled using two service centers for each NIC, according to the model of a Web server node that Bianchini et al. propose in [38]. Figure 5.2 shows the queue model: **NIC0i** and **NIC1o** are for delivering client-to-server packets; **NIC1i** and **NIC0o** are for delivering server-to-client packets.

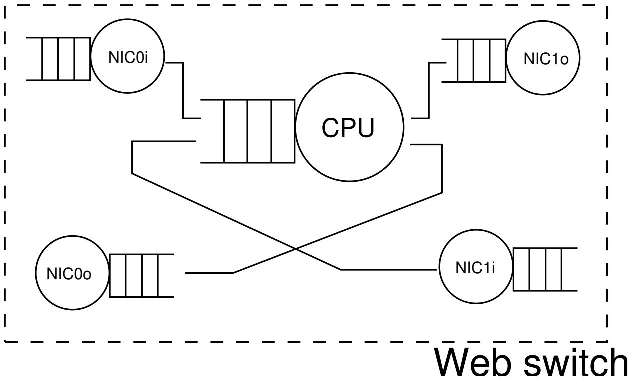


Figure 5.2: Queueing network model of a two-way Web switch.

5.1.2 Web server model

The Web server model that we propose is shown in Figure 5.1. The system components modeled are: the CPU, the disk and the memory (we suppose that the network interface card of the server does not represent a system bottleneck). We suppose that when a Web request is assigned to a Web server the service center are visited as it follows:

1. The **CPU** parses the request and fetches the response document from the Web server cache (main memory) or from the disk in the case of static request and from the back-end nodes for dynamic requests.

2a. **Main memory or disk** is used with probability P_{hit}, while $P_{miss} = 1 - P_{hit}$ is the Web server cache miss probability.

2b. **Back-end** nodes are used with a probability P_{dyn} that corresponds to the probability that a client request is for a dynamic document that typically may not be cached.

3. The **CPU** is used to produce the response back to the client.

5.1.3 Back-end server model

We introduce back-end nodes to simulate real component such as application servers, database servers or generally enterprise servers. These are complex systems and an accurate modeling is outside the scope of this thesis. Each of these nodes is modelled as a single service center model.

5.1.4 Web client model

Looking at the Web users, we consider an open model: Web clients enter the system with an arrival rate defined by some probability distribution function (in Chapter 7 we will give details about the workload model). Users ask to the Web cluster a randomly generated number of services (i.e. one or more Web page(s)), and leave the system until all Web requests issued are satisfied or the simulation time is over.

If we consider a real Internet scenario not all Web clients are equal in term of access speed. This heterogeneity has different nature: for example a powerful client node connected trough a slow network link result in a slow Web client and vice versa. In this thesis we introduce three categories of Web clients: *Fast*, *Medium* and *Slow*. As shown in Table 5.1, the speed of a Web client is given by the node power and interconnection bandwidth.

Client node	Network connection	Web client
Fast	Slow	Slow
Fast	Medium	Medium
Fast	Fast	Fast
Slow	Fast	Slow
Slow	Medium	Slow
Slow	Slow	Slow
Medium	Slow	Slow
Medium	Fast	Medium
Medium	Medium	Medium

Table 5.1: Client speed classification.

5.2 WebSim: A Web cluster simulation tool

The queuing network model is evaluated through WebSim, a discrete-event driven simulator. WebSim implements the system model through the CSIM18 package [88], a library of routines for use in constructing process oriented, discrete-event simulation model. Figure 5.3 shows an high level view of the WebSim software structure.

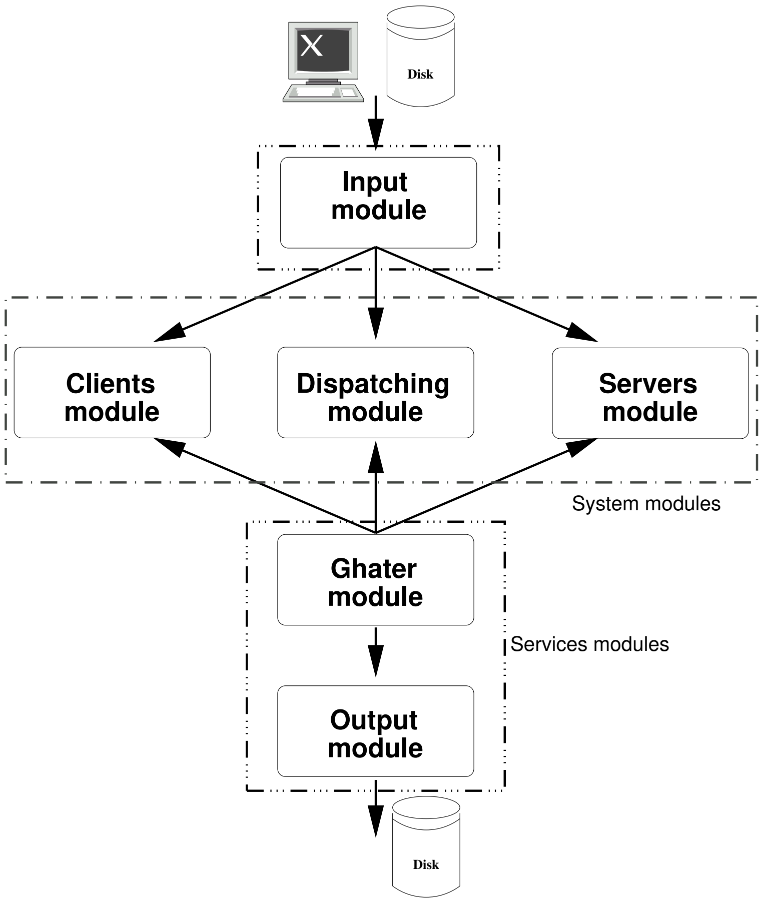


Figure 5.3: Software modules of WebSim simulator.

There are three software modules that define the system behavior (*Clients module*, *Dispatching module* and *Server nodes module*) and other software modules which implement simulation services such as input/output and statistics gathering (*Input module*, *Output module* and *Gather module*).

5.2.1 Client module

This module implements the client side of the system. Each client is a CSIM process activated at a time t_i, a random variable defined by some distribution function that models the client interarrival time.
After activation the Web client initializes itself and it computes session parameters. First is generated the *number of Web page* that the client requests in a *Web session*, i.e. the entire period of connection to the site. For each Web page is generated the *number of embedded objects* and their type that is, if objects are static or are dynamically generated. We define *dynamic Web page*, a Web page with at least a dynamic object. Once defined the section parameters, the Web client enters the system and generates the first request of connection to the Web switch of the Web cluster.

Client-server interactions are modeled at the details of TCP connections, including packets and ACK signals. Because in our workload, defined in the next section, we will consider secure connections, we model all details of communication and Web server overhead, due to key material exchange, server authentication, encryption and description of user data.

Then the CSIM process sends a connection request message[1] to the Web switch and wait for response. The Web switch, selected a Web server node, send to it a CSIM message to simulate the Web switch-Web server communication mechanism and to activate an HTTP CSIM process, which will serve the client request. The Web server send a message to the client to simulate the HTTP response. The Web client process is resumed by the Web Server response message, then it may submit the request for an embedded object of the same Web page or wait a *user think time* T_{tt}, i.e. the time required to analyze the requested page and decide about to leave or surf again the site asking for a new Web page.

To model Slow, Medium and Fast client we introduce a delay between an embedded object request and the next. This delay is proportional to the speed of the client. Another parameter is the fraction of client in each category. We define P_{Slow}, P_{Medium} and P_{Fast} (where $P_{Slow} + P_{Medium} + P_{Fast} = 1$) the probability that a request is from a Slow, Medium or Fast Web client respectively.

We do not consider client software that issues more requests concurrently like

[1]CSIM provides an intra process communication feature based on message exchange and mailboxes. When we refer to message exchange we intend a CSIM message.

Netscape[REF], because this technique improves the performance of a Web client but do not impact Web server performance.

Client-server interactions are modeled at the details of TCP connections, including packets and ACK signals. Because in our workload we consider also secure connections, we model all details of communication and Web server overhead, due to key material exchange, server authentication, encryption and description of user data.

5.2.2 Server module

The server module implements the Web server and back-end node model. Each Web server component, i.e. CPU, disk, and memory, and the back-end node is a CSIM resource having its own queuing system that allows for requests to wait. To be realistic, these hardware components are parameterized in detail. For example, the disk is defined by transfer rate, controller delay, seek time and RPM; the memory is parameterized with its capacity and transfer rate.

When a Web server receives by the Web switch an HTTP request, a CSIM process simulating an http daemon is activated.

The Web server software is modeled as an Apache-like server, where an HTTP daemon, activated at the system initialization, waits for connection requests. This means that when the Web switch sends a request message to a Web server it forks a new HTTP process (CSIM process) designed to serve it. The activated HTTP process requests a service to the CPU, to simulate the HTTP request parsing action. After the process is resumed it uses the disk, the memory or a back-end node. The memory, introduced to simulate the Web server cache is managed with a LRU cache replacement algorithm. We model both HTTP/1.0 and HTTP/1.1 protocols. With HTTP/1.0, a new HTTP process is forked to serve each Web page component that is, the HTML file and all the embedded objects. With the HTTP/1.1 protocol, the same HTTP process serves the HTML file and all embedded objects.

5.2.3 Dispatcher module

This module implements all the routing mechanisms and dispatching algorithms. When the Web switch receives a connection request, it may or not accept it, this because an admission control mechanism could be applied to avoid system overload. We suppose that incoming requests are all accepted. Once a Web server is chosen, the Web switch sends to it a message containing information about the client process and to replay and about the requested document.

Because some of the dispatching algorithms are based on server state information, the Web switch collects load state information from the Web servers. To this purpose a CSIM process every T_{get} seconds stores load state information about each Web server such as number of active process, server response time, CPU and disk utilization.

Class	**Input parameters**
Workload Parameters	Web client arrival rate
	Request per session (Web pages)
	Objects per request
	Object size (HTML file and embedded objects)
	Number of Web objects
	Frequency of dynamic requests (P_{dyn})
	Web client nature (P_{Slow}, P_{Medium} and P_{Fast})
System Parameters	Number of Web servers
	Number of back-end nodes
	Memory capacity
	Memory transfer rate
	Disk transfer rate
	Seek time
	Controller delay
	RPM
	Back-end node service time
	NIC(s) bandwidth
	HTTP protocol version
	Cache replacement algorithm
	Back-end selection strategy
Dispatcher Parameters	Number of Web switch
	Routing level (4/7 layer)
	Dispatching policy
	Web switch gather time (Tget)
	Server load metrics

Table 5.2: WebSim Input Parameters.

5.2.4 Others modules

The service modules implement functionalities related to the simulator such as providing the input parameters, collecting simulation statistics and producing the simulation reports. Now we describe more in detail this functionality:

Input module provides to system modules parameters needed to the system initialization and configuration. This data could be provided to the WebSim through an input file or through a command line.

WebSim inputs can be divided into three main classes: *Workload parameters* needed to initialize the Client module, *System parameters* needed to initialize and configure the Web switch, Web servers nodes and back-end nodes, and *Dispatching algorithms parameters* which configure the dispatching algorithms. Table 5.2 summarizes the main WebSim inputs with a short description.

Gather module is the WebSim "informer". It collects in CSIM tables and histograms the system status, eventually elaborates some performance value to compute performance metrics and, at the end of the simulation, it reports all data to the output module.

Output module produces the simulation reports on the basis of the values provided by the Gather function. Table 5.3 summarizes the main WebSim outputs with a short description.

Output parameters	**Metrics**
Throughput	connections/sec.
Throughput	byte/sec.
Throughput	hits/sec.
Throughput	Web page/sec.
Server(s) Hit response time	seconds
Cluster Hit response time	seconds
Server(s) Page response time	seconds
Cluster Page response time	seconds
Session response time	seconds
System (single node) utilization	[0,1]
CPU(s) utilization	[0,1]
Disk(s) utilization	[0,1]
Memory(s) utilization	[0,1]
Back-end(s) utilization	[0,1]

Table 5.3: WebSim Output Statistics.

Chapter 6

ClubWeb prototype

Any performance study carried on through a simulation tool, cannot consider all factors that affect a real system, just to give some real examples tuning of operating system parameters and process concurrency limits, network-kernel interactions. For these reason, we decided to realize a Web cluster prototype that implements all dispatching algorithms and most mechanisms discussed in this thesis.

The name of the prototype is **Clu**ster-**b**ased **Web** system (**ClubWeb**)

6.1 Architecture of the ClubWeb prototype

ClubWeb is a Web cluster system based on off-the-shelf hardware and software components. It implements a layer-4 and a layer-7 Web switch that supports all class of dispatching algorithms presented in Chapter 4. Figure 6.1 shows an high level view of the ClubWeb architecture.

The Web cluster consists of a Web switch node, connected to the Web servers and the back-end nodes through a high speed LAN (Fast-Ethernet). The distributed architecture of the cluster is hidden to the HTTP clients through a unique Virtual IP (VIP) address.

At the Web switch node, the dispatching functionalities are implemented by a module working at the kernel or at the application level, for a layer-4 or a layer-7 Web switch, respectively.

Each Web server node runs the HTTP daemon and the middle-ware software needed to interact with the back-end nodes. Back-end nodes run an application software, for example a data base management system or a business logic to interact with a legacy system. The adopted middleware software depends on two main factors: the

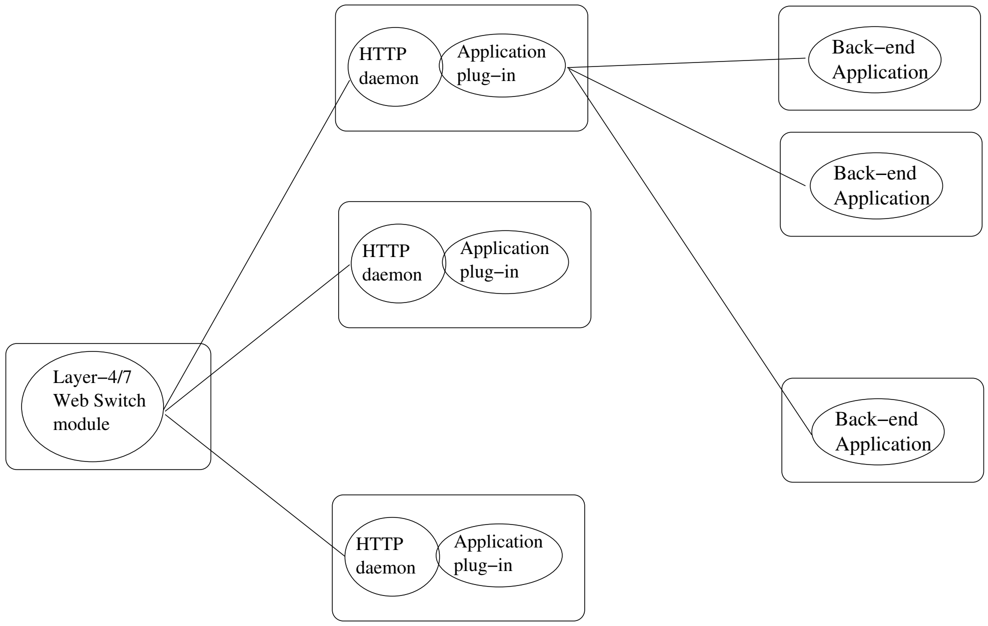


Figure 6.1: Architecture of the ClubWeb prototype.

technology used to generate the dynamic pages and the technology used to implement the application running on the back-end nodes.

ClubWeb uses Apache 1.3.12 as the Web server software, PHP 3.0.17 as dynamic page engine and MySql as a data base management system at the back-end nodes.

The experimental results refer to a platform where the Web switch node, the Web server nodes and the back-end nodes are Dual PentiumIII-833Mhz PC with 512MB of memory and with a 3Com 3C905C 100bTX network interface. The operating system of all node of the cluster the cluster nodes is the Linux O.S., kernel release 2.4.17.

6.2 Layer-4 Web Switch

Different mechanisms were proposed to implement a layer-4 Web switch. The most popular are the Packet single-rewriting, Packet Tunneling and the Packet Forwarding (or Direct Routing) techniques. The comparison analysis carried out in Chapter 3 led us to choose the Packet Forwarding mechanism because it limits the overheads of packet rewriting and two-way architecture. As the Web switch processes only inbound packets at MAC layer, expensive checksum recalculations are avoided. This Web switch solution is able to sustain peak loads of thousands of connections per seconds hence often the cluster scalability is limited by the capacity of the network link that connects the Web cluster to Internet.

The ClubWeb layer-4 Web switch module is based on the Linux Virtual Server (LVS) [82], a kernel loadable module that permits as to configure a Server node as a router for requests directed to some predefined network service, such as HTTP, FTP, SSH, TELNET.

The ClubWeb prototype results as one server to an outside client. The HTTP and back-end servers are under the control of a Web switch which runs a Linux kernel patched to include the *ipvs* code. The ipvs code running on the dispatcher is the essential feature of the ClubWeb prototype, although other user level code can be used to manage the cluster. Figure 6.2 shows an high level view of the Web switch architecture.

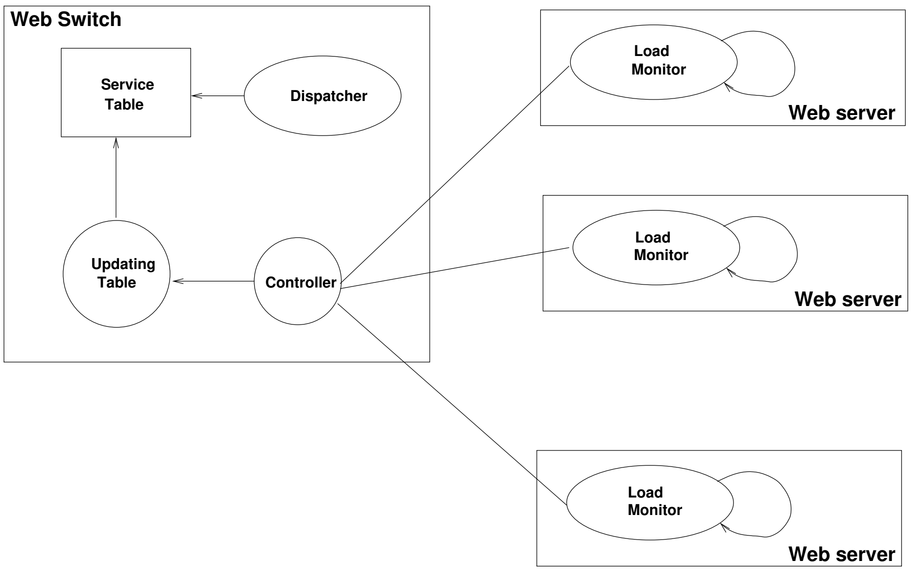


Figure 6.2: Modules of the layer-4 Web switch.

When a packet arrives the *Dispatcher module* lookups the *connection table* to decide whether the incoming packet is the first of a new connection or one referring to an already established connection. When the first packet of a connection arrives, the Web switch apply a dispatching algorithm to a Web server and updates the connection table, then forwards the packet to the designed server. All the other packets of this connection are directly forwarded to the designed server on the basis of the MAC address stored on the connection table. Each Web server receives packets as directly coming from the client, and knows the client IP address to which it has to send the response without passing through the Web switch.

Figure 6.3 shows the architecture of the Web switch.

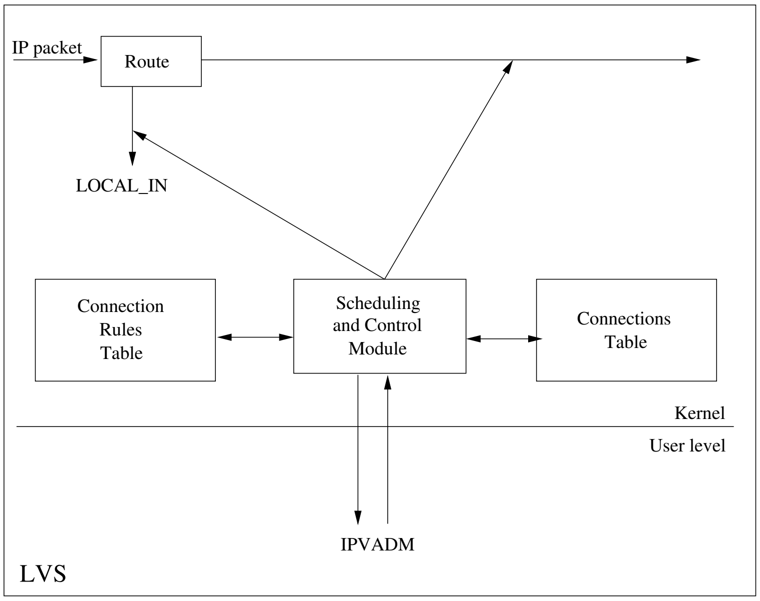


Figure 6.3: Web switch detailed architecture

The Dispatching module is the main component of the LVS. The incoming IP packets are intercepted in two points of the route (into the kernel network module) to execute the rewriting and encapsulation operations. The Dispatching module checks into the *Rewriting Rule table* if the incoming packet is for a new connection. If the lookup fails, this module identifies, into the Connection table, to which established connection of the incoming packet.

The Dispatching module contains the *Service table*, that stores all the information related to: the VIP address, the network services associated to the VIP, the port numbers that identify the network services, the Web server associated to the VIP and the servers associated to other services. An example of Service table is shown in Figure 6.4.

The Connection table is an hash table designed to store the information related to millions of concurrent connections. Each record of the table requires 128 bytes of memory. Hence a Web switch with 256 Mbyte dedicated to connection and dispatching operation is able to manage up to 2 millions of concurrent connections.

The Web switch of the ClubWeb prototype implements both state-blind and state-aware dispatching algorithms. Therefore need mechanisms to collect load information on the server nodes and to gather it at the Web switch. In particular, each Web server

VIP	PORT	REAL SERVER	REAL PORT
VIP1	P1	RIP1 RIP2 RIP3 RIP4	P1
VIP2	P2	RIP6 RIP3 RIP4 RIP5	P2

Figure 6.4: The Service Table

runs a *Load Monitor*, collects load state information at the application level.

The Load Monitor integrates several operating system performance evaluation information from the *proc file system*, such as: disk, CPU and memory utilization, number of active connections for each service provided by the server.

Each node collects load state information at different sampling intervals, and summarizes them if required. For example, calculate the arithmetic or weighted mean value of the last samples. Each Web server node periodically sends the load state information to the Controller process on the Web switch. The communication between each Load monitor and the Collector module is carried on over a direct UDP connection on a dedicated channel (one channel for each Web server). The frequency of messages received by the Collector over these channels is also used as a hearth-bit signal. Web servers not sending information are considered unreachable or overloaded, and are excluded from the pool of the possible choices of the Dispatcher module.

The Collector elaborates the information provided by the Web servers, to allow the application of some server state-aware dispatching algorithms. For example, in the case of the Weighted Round-Robin policy it periodically updates the weights associated to each Web server.

6.3 Layer-7 Web Switch

The first release of the ClubWeb prototype implements the TCP gateway (also known as reverse proxy or surrogate) as a layer-7 mechanism, although we are testing a new version using the more efficient TCP hand-off technique. In the present version, a proxy-like process runs on the Web switch at the application layer and mediates the communication between the client and the server on a two-way basis.

When a new connection request arrives, the Web switch establishes the TCP connection with the client and waits for the HTTP request. In such a way, it can apply some content-aware dispatching algorithms in the selection of the target server. Once decided the Web server, the Web switch forwards the client request to it. When the response arrives from the Web server, the Web switch forwards it to the client through the client-switch TCP connection.

We implemented the TCP gateway mechanism through the Apache HTTP server software, that was adequate to test all layer-7 dispatching algorithms without modifications at the kernel level. There is no doubt that a system that has to manage thousands of requests per seconds should work at the kernel level, but many commercial systems still use the TCP gateway mechanism that is more portable and easy to implement (see Section 3.2.3).

Figure 6.5 shows the architecture of the Web switch software and its interactions with the Web servers.

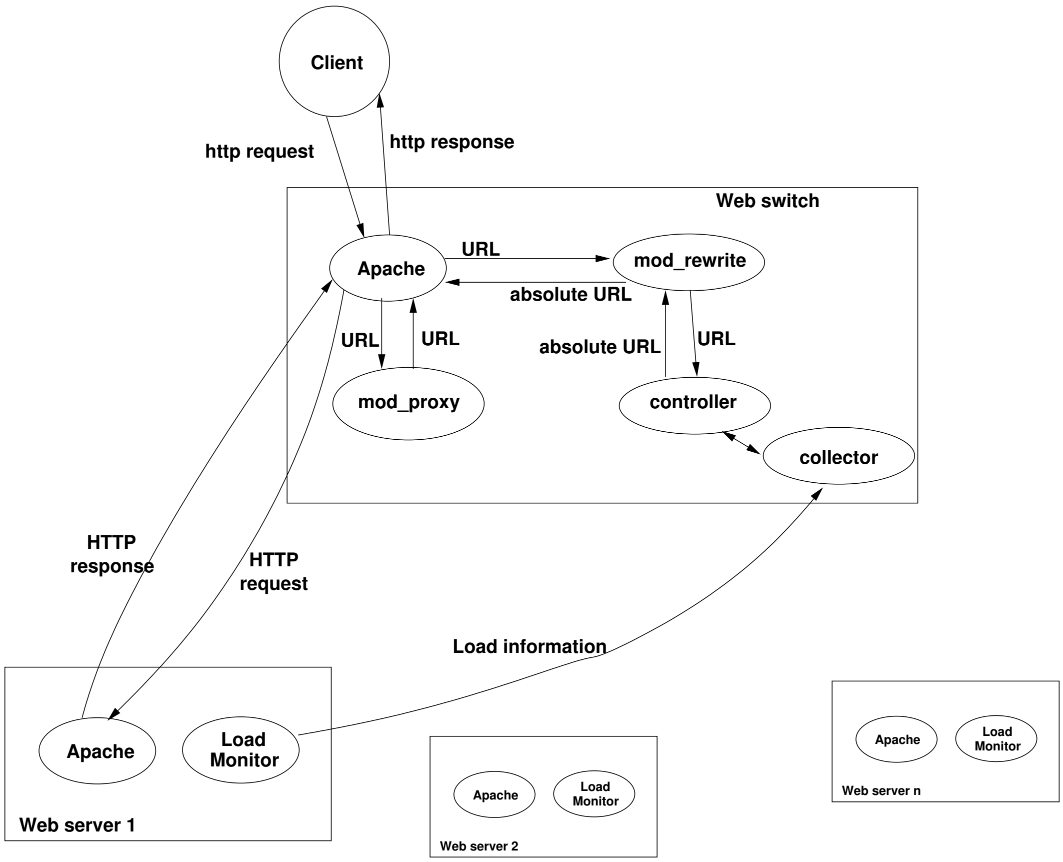


Figure 6.5: Architecture of the layer-7 Web switch software and interactions with the Web servers.

The software on the Web switch is basically configured to work as a reverse proxy through the modules *mod_proxy* and *mod_rewrite* [9]. They provide the proxy serving and URL rewrite engine mechanism, respectively. Below, we give some details about the functionality of these two modules.

mod_proxy implements the proxy and reverse proxy functionalities in the Web switch for the protocols FTP, CONNECT(SSL), HTTP/0.9, HTTP/1.0 and HTTP/1.1. The main tasks executed by the mod_proxy module are:

1. to build a new URL on the basis of the information provided by the mod_rewrite that is, the name of the designed server;
2. to forward, as a *proxy request*, the incoming request to the appropriate Web server;
3. to modify the header of the HTTP response provided by the Web server, by changing all the Web server name references.

mod_rewrite provides all services related to the URL rewriting mechanism. This module provides a configuration directive that permits to realize the following tasks:

1. to intercept an incoming URL whose format matches with a known pattern (this functionality is fundamental to realize the content-aware routing).
2. to specify the rewriting mode, that may be static (if we use a lookup table) or dynamic (if we use a software module that communicates with the Apache Web server through the stdin/stdout file handle). The dynamic rewriting mode guarantees the highest level of configurability to the Web switch. We use this method because it permits to implement the dispatching algorithms in a extern module that is able to communicate with the load monitors running on the Web server nodes. Another feature provided by an external module is the possibility to change the content-aware dispatching policy at run time. This feature is important to implement some adaptive algorithm, although no example has been yet proposed in literature.
3. to specify, through a dedicated flag, if the rewrite packet needs to be forwarded directly to the Web server or if needs to be rewritten again.

The content-aware dispatching algorithms are implemented in the *Dispatcher* module of the Web switch. This is a C module that communicates with the rewriting engine, which is provided by mod_rewrite over its *stdin* and *stdout* file handles.

For each map function lookup, the Dispatcher module receives the key to lookup as a string on *stdin*. After that, it has to return the looked-up value as a string on *stdout*.

The layer-7 Web switch of the ClubWeb prototype implements both client state-aware algorithms and server state-aware dispatching algorithms, such as WRR and LARD (Sections 4.2.3 and 4.3). For these latter policies we had to implement a mechanism to collect load information at the Web servers and to gather these data at the Web switch. Indeed, each Web server runs a *Load Monitor*, that is an application level module dedicated to collect, elaborate and send server load state information to *Collector* module of the Web switch.

The Load Monitor integrates operating system performance evaluation tools (such as *iostat*) and custom programs that get information from the *proc file system*. It is a flexible module that, depending on the ClubWeb administrator choice, can collect various system load information, such as CPU, disk and memory utilization, number of active connections and combinations of them.

The Load Monitor of a Web server periodically sends the load state information to the *Collector* module running on the Web switch. The communication between each Load monitor and the Collector module is carried on over a direct UDP connection on a dedicated channel (one channel for each Web server). The frequency of messages received by the Collector over these channels is also used as a hearth-bit signal. Web servers not sending information are considered unreachable or overloaded, and are excluded from the pool of the possible choices of the Dispatcher module.

Chapter 7

Workload model

Special attention has been devoted to the workload model that incorporates the most recent results on Web load characterization.

World Wide Web is become a complex media through which a large variety of services are provided. Hence, it is impossible to define a workload that models the behavior of all the possible Web services. A common technique used to reproduce a specific Web site/service behavior is trace-based simulation. As the goal of this thesis is to evaluate the system performance for severs workload scenarios, we preferred to refer to distribution-driven simulation. The purpose is to define a workload that can reproduce the performance impact of the most common Web services. We consider three main classes of load:

Static Web services. Requests for HTML pages with some embedded objects. Typically, this load has a low impact on Web server components. Only requests for very large files are disk and network bound.

Dynamic Web services. Requests for HTML pages, where objects are dynamically generated through Web and back-end server interactions. Typically, these requests are CPU and/or disk bound.

Secure Web services. Requests for a dynamic page over a secure connection. Typically, these services are CPU bound because of overheads to setup a secure connection and to execute cryptography algorithms.

The most complex types of Web services require the generation of dynamic data, transportation of sensible information, and integration with Legacy/Enterprise systems: common examples are e-commerce, e-business, on-line auction, etc.

Without enter into the details of these Web-based applications, the workload models typically use a mix of static, dynamic and secure workload as a load of a Web commerce site.

7.1 Static load

The workload model of static requests incorporates the most recent results on Web load characterization. The high variability and self-similar nature of Web access load is modeled through heavy-tailed distributions such as Pareto and lognormal distributions [13, 21, 53].

A Web client enters the system with an arrival rate that follows an exponential distribution. As mentioned in Chapter 5, a client process populates the system for a *Web session*. The number of *requests* per Web session that a client will request is modeled according to the inverse Gaussian distribution [100]. For each *embedded objects* of a Web page, the Web client issues a request to the Web cluster. The number of *embedded objects* per request, including the base HTML file, is also obtained from a Pareto distribution [20]. Once received a Web page, the Web client may generate a new request after a *user think time* that follows a Pareto distribution [20, 100].

Static Web pages, are composed of an HTML file, with size distribution obtained from a hybrid function, where the body follows a lognormal distribution, while the tail is given by a heavy-tailed Pareto distribution [13, 20, 100]. The size distribution of embedded objects, instead, is obtained from the lognormal distribution [20].

Table 7.1 summarizes the probability mass function (PMF), the range, and the parameters' value we use in the workload model.

Category	*Distribution*	*PMF*
Requests per session	Inverse Gaussian	$\sqrt{\frac{\lambda}{2\pi x^3}} e^{\frac{-\lambda(x-\mu)^2}{2\mu^2 x}}$
User think time	Pareto	$\alpha k^{\alpha} x^{-\alpha-1}$
Objects per request	Pareto	$\alpha k^{\alpha} x^{-\alpha-1}$
HTML object size	Lognormal Pareto	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(ln\, x-\mu)^2}{2\sigma^2}}$ $\alpha k^{\alpha} x^{-\alpha-1}$
Embedded object size	Lognormal	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(ln\, x-\mu)^2}{2\sigma^2}}$

Table 7.1: Workload model.

7.2 Dynamic load

A dynamic Web page is composed of an HTML file, some static embedded object and some dynamic object. A dynamic object is a file which is generated through a query to a back-end server. We suppose that the file size of dynamic object follows the same distribution of the static ones, the number of dynamic objects per page instead follows a geometric distribution.

A dynamic request includes all overheads of a static request and overheads due to back-end server computation to generate the dynamic objects. We do not specific the nature of the generated object, it may be a string of few bytes resulting from an exhaustive database search, or a big image retrieved from a small repository. We focus on the performance impact of dynamic requests on the back-end server.

We start from the request classification given by the *Transaction Processing Performance Council* (TPC) consortium in the *TPC-W* benchmark [24], the most authoritative e-commerce benchmark.

The TPC-W workload simulates the activity of a retail store Web site, where costumers can browse and buy books. TPC-W user requests are summarized into 14 queries, organized in two main classes: browsing requests and order requests. The browsing category includes even static requests, such as home page request, and dynamic but cachable requests, such as best seller and new product requests. These cases are already included in the static workload model (Section7.1).

We observe that the the TPC-W can be summarized in three categories, each one with a different performance impact on system performance. So we have:

Light-intensive requests include browsing operations like shopping chart management, search action, order inquiry, product details.

Middle-intensive requests include order operations, such as order display, buy requests, best sellers queries.

Intensive requests include critical operation, such as buy confirm and administration account confirm.

Some of Middle-intensive and Intensive requests are carried out over secure connections. In our model we separate the overhead due to the type of query from the secure connection load, modelled in Section 7.3.

Each class of requests, is characterized by an exponential service time on back-end nodes with mean value μ_{Light}, μ_{Middle} and $\mu_{Intensive}$. Because service times are strictly dependent on the hardware and software technology, some sensitivity analysis is done on these parameters.

We model the composition of dynamic load, through P_{Light}, P_{Middle} and $P_{Intensive}$ probabilities of occurrence of Light-intensive, Middle-intensive and Intensive requests, respectively. Obviously, $P_{Light}+P_{Middle}+P_{Intensive}=1$ and, if P_{dyn} is the probability of dynamic requests, $P_{dyn}*P_{Light}$ is the frequency of dynamic Light-intensive requests, and so on for the other types of requests. Values for P_{Light}, P_{Middle} and $P_{Intensive}$ parameters are extracted from the TPC-W benchmark.

Category	*Mean Service Time*	*Frequency*
Light Intensive	μ_{Light}	P_{Light}
Meddle Intensive	μ_{Middle}	P_{Middle}
Intensive	$\mu_{Intensive}$	$P_{Intensive}$

Table 7.2: Dynamic workload model.

7.3 Secure load

Secure transactions among clients and Web servers involve some secure connection protocol. The standard proposed by the IETF is the TLS (Transport Layer Security) defined by the RFC2246 [60]. TLS protocol is derived by the standardization of the SSL (Secure Socket Layer) protocol implemented by the Netscape Communication and today used by 40 millions of users in the world. Hence the model of the SSL protocol includes main CPU and transmission overheads due to SSL interactions, such as key material negotiation, server authentication, and encryption and decryption of key material and Web information. The CPU service time consists of encryption of a server secret key with a public key encryption algorithm, such as RSA, computation of Message Authentication Code through a hash function, such as MD5 or SHA, and data encryption through a symmetric key algorithm, such as DES or Triple-DES.

Most *CPU overhead* is due to data encryption (for large size files), and public key encryption algorithm (RSA algorithm), that is required at least once for each client session, when the client has to authenticate the server.

The *transmission overhead* is due to the server certificate sent by the server to the client, the server hello and close message, and the SSL record header.

The throughput of encryption algorithms depend on characteristics of the system under investigation: for example, if we use a dedicated device for encryption and decryption, the CPU will not be overloaded, or on the other hand, a system with a slow CPU or NIC will be completely overloaded by a secure transfer protocols. A server with a Pentium II/III processor and a 100Mbit Ethernet card will provide medium performance.

The CPU overhead is characterized through the en/decryption algorithms and hash function throughput (Kbps). On the other hand, the transmission overhead is simply characterized by increasing the size (in byte) of the transmitted connection message or Web document.

Table **??** summarizes the *secure* workload parameters: Thr_{RSA}, Thr_{3DES} and Thr_{MD5} denote the throughput of encryption algorithms and hash functions, respectively. S_{cert}, S_{hello} and S_{rec} are the transmission overhead introduced by the server certificate, the server hello and close message, and the SSL record header, respectively.

Parameters	*Value*
Throughput	Thr_{RSA}
	Thr_{3DES}
	Thr_{MD5}
Message Overhead	S_{cert}
	S_{hello}
	S_{rec}

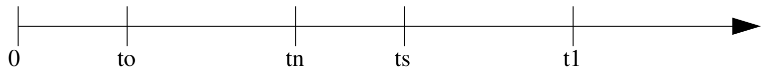
Table 7.3: Secure workload model.

7.4 Performance metrics

Once defined the system and the workload model, the next step is to define the most appropriate *performance indexes* to evaluate cluster-based Web systems. In this section we give a description of the possible performance indexes for the system under investigation and the actor (users, service provider, system administrator) which can benefit from the optimization of such indexes. We consider the four main performance indexes: *response time*, *latency time*, *throughput* and *utilization*.

Response time is defined as the time experienced by a Web client to complete the entire request or, in alternative, as the time to obtain the first response packet. Figure 7.1 shows a general example of response time, without detailing of the system service time and network delays.

Because of the nature of the response time, the service provider could control only a small percentage of such index: once optimized the system service time,

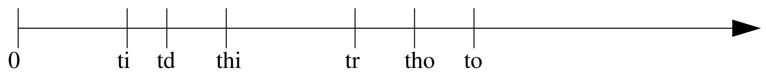


tn – to = request delay
ts – tn = system service time
t1 – ts = response delay
t1 – to = response time

Figure 7.1: Request response time.

the service provider could only decide to apply some mechanism or technology to reduce the network delay. However factors such as slow client connection can not be controlled by the content provider.

Latency time is the time needed by the system to complete a Web object (or document) request. Network delays are excluded from the latency time metric.
Latency time could be measured for the entire Web cluster, or for each system components, such as the Web switch, Web server and Back-end nodes.
This index is under the full control of the service provider, and its optimization guarantees an increase of the number of user requests processed in a time unit, i.e. the system throughput. Figure 7.2 shows a typical composition of the latency time measured at the Web cluster.
The *dispatcher service time* is the time needed by the Web switch to chose a Web server and to route the incoming request; the *httpd parsing time* is the time needed by the HTTP process to parse the incoming HTTP request; the *request fetching time* is the time needed to retrieve (from disk or memory) or to produce (from back-end server) the requested Web object; the *sending time* is the time needed by the HTTP process to produce the HTTP response; finally the response is packeted and sent in a *sending time*.



td – ti = dispatcher service time
thi – td = httpd parsing time
tr – thi = request fetching time
tho – tr = httpd packing time
to – tho = sending time
to – ti = latency time

Figure 7.2: Request latency time.

We also propose a cumulative measure of the latency time, named *Page latency time*, that is the time needed by the system to complete a Web page request: $T_{page} = \sum_{i=1}^{n\_obj} T_i$ where n_obj is the number of objects embedded in a Web page, and T_i is the latency time of the object i.

Throughput is defined as *quantities* computed by the system or resource in a time unit (e.g. second).
This index is controlled by the service provider and must be maximized to augment the number of satisfied users and, eventually, to increase the Web site profit. Quantities of interest may be: hit, file, (K)bytes, connections, Web pages. Therefore the system throughput can be expressed in many forms, such as:

- number of hits completed in a time unit (e.g. second)
- number of dynamic hits per second
- number of static hits per second
- number of files served per second
- number of (K)bytes served per second
- number of (HTTP or TCP) connections established per second
- number of Web pages served per second

Utilization is the fraction of a time interval T that a *resource* or *system* is busy. As the system is a complex multi-component Web cluster, the resources could be CPU, disk, memory, NIC and Back-end node. Figure 7.3 shows an example of resource utilization: if we observe a resource, for example the CPU, for the time interval T and we find that in the subinterval Ta, Tb and Tc the CPU is busy in serving a client request or performing an operating system task, the resource utilization is $U = \frac{T}{Ta+Tb+Tc}$. Obviously, we have that $0 \leq U \leq 1$.

The utilization of a multi-resource system may be measured in many ways. We propose that the system utilization, for example the Web server node utilization is given by the $U_{server} = max\{U_{cpu}, U_{disk}, U_{memory}\}$, because in a real scenario if the CPU, memory or disk is overloaded the system might be considered overloaded as well. Hence, we give the following definitions of system utilization: *the*

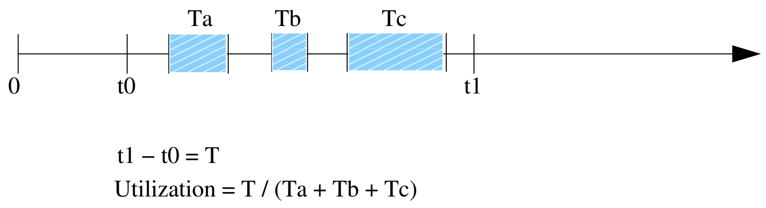


Figure 7.3: Resource utilization.

fraction of a time interval that at least one of the system resources (i.e. CPU, disk and memory) is busy at time t.

Table 7.4 resumes the characteristics of the considered performance indexes.

Performance Index	*Point of measure*	*Service provider control*	*Optimization*
Response time	Web client	Partial	Minimize
Latency time	Web cluster bounds Web server node(s)	Total	Minimize
Throughput	Web cluster Web server node(s)	Total	Maximize
Utilization	Web cluster Web server node(s) Single component(s)	Total	Maximize

Table 7.4: Performance Indexes.

7.4.1 Statistical measures

Once defined the performance indexes, we need some metrics to extrapolate a single or a set of values that are representative for that index. Common statistical measures of a set of sampled data are: *sample mean*, *sample standard deviation*, *X-percentile* and *cumulative distribution function*.

Sample mean and standard deviation are representative measures of a set of sampled data following a standard probability mass function (e.g. normal, Poisson, etc...). In our workload, all parameters have an heavy-tail distribution that is, a probability distribution characterized by an infinite standard deviation. For this type of distribution, the *mean value* is not representative, hence the most representative statistical metrics are the X-percentiles and the cumulative distribution functions.

X-percentile of a random variable x is defined as $P(x <= X) = Y$. It gives a measure of which is the degree of satisfaction with respect to a performance

index. For example, if the X-percentile of the page delay is less than T seconds, this means that in the $X\%$ of the system observation the page delay is less than T seconds or alternatively, that the users experiment a page delay greater than T seconds with a probability $1 - X/100$.
Typical measure of interest are the 90 or 95-percentile.

Cumulative distribution is a function (cdf) that give the probability that a performance index takes a value less than or equal to x. It is formally defined by $F(x) = P(X \leq x)$.

Unbalancing Factor(UF) is derived by the *Load Balance Metric* (LBM) proposed in [29]. LBM is the weighted average value of the peak-to-mean ratio that is defined as $\frac{peak\_load_j}{(\sum_{i=1}^{n} load_{i,j})/n}$, where $peak\_load_j$ is the peak load at the j^{th} sampling periods and $load_{i,j}$ is the load of server i at the same time. The definition of LBM for a system with n servers and m sampling periods is:

$$LBM = \frac{\sum_{j=1}^{m} \left(\frac{peak\_load_j}{(\sum_{i=1}^{n} load_{i,j})/n} * \frac{\sum_{i=1}^{n} load_{i,j}}{n} \right)}{\sum_{j=1}^{m} \sum_{i=1}^{n} load_{i,j}/n}. \tag{7.1}$$

This measure shows the ability of the dispatching policies to share the load through the servers of the Web cluster. The load index may be the number of requests being served concurrently (LBM-Hit) or the time to serve 1 KByte request (LBM-Byte). In our experiments, these load measures are sampled at each server every 10 seconds.
The *Unbalance Factor (UF)* is the percentage difference between the LBM value and the best LBM value which is achieved when the load is perfectly balanced among the servers. By definition, LBM ranges from 1 and the number of servers n, so we have

$$UF = \frac{LBM - 1}{n - 1}. \tag{7.2}$$

This percentage index gives a more immediate measure to compare the performance results of different dispatching policies.

7.5 Benchmark tool

We found useful to complete a performance study carry out through simulation with measurements on a real prototype of the Web cluster system.

This analysis requires the implementation of the workload model described in the previous sections through a benchmarking tool.

We started from the WebStone benchmark [89] and modified it in many ways.

WebStone creates a load on a Web server system by emulating the activity of multiple clients, which are named *Web clients* and which can be thought of as users, Web browsers, or other software that retrieves files from a Web server. WebStone uses multiple Web clients running on one or more computers. A master process, named *Web master*, coordinates Web clients, starting them on remote machines, and collects cumulative statistics when the run time is over.

For the purpose of our analysis, WebStone presents two main problems related to the client behavior and performance metrics.

- Webstone models a closed system. In the Web client life cycle, there is not the concept of user session; a Web client issues requests to the Web system as soon as possible that is, a new Web request is sent immediately, after the response to the previous one is received. Although in a more realistic scenario, the client requests a set of Web objects (the HTML file and the embedded objects) after that, the human user "parses" the received Web page and eventually decides to issue another request to the Web site.

- The performance metrics reported by WebStone are based only on mean values. It does not consider X-percentiles nor Cumulative Distribution Functions that are fundamentals for performance analysis under heavy-tailed distributions.

To solve these problems, we modify the WebStone benchmark by introducing the user think time in Web clients life cycle, and we cumulate values of different performance indexes in relatives density mass functions. In this way we are able to calculate also the X-percentiles and cumulative distributions. We also give the possibility to pass from a closed loop model to an open system where client requests are issued independently of the response from the servers.

Chapter 8

Content-blind dispatching policies: A performance study

In this Chapter we report the main results of a performance study on content-blind dispatching algorithms. In particular, we focus on Random (RAN) Round-Robin (RR), that are server state-blind algorithms, and Weighted Round Robin (WRR) that is a server state aware policy.

We refer to the simulation model proposed in Chapter 5 and to the Unbalancing Factor measure described in Section 7.4.1. Table 8.1 outlines the system and workload parameters.

In the scalability analysis our goal is to demonstrate which is the system behavior when the offered load increases and how Web clients perceive the correct tuning of the architectural parameters. We also use the 90-percentile of Page delay as an additional performance metric.

8.1 Performance analysis

We consider two main classes of services for a Web site: (1) Web clusters providing HTML pages with embedded objects and large files such as multimedia and software files; (2) Web Clusters providing even dynamic pages that may require intensive CPU and/or database operations.

8.1.1 Static contents scenario

In the first set of experiments we evaluate the impact of dispatching policies when the Web site provides only static (even if potentially large) objects. We analyze

	Parameter	*Value*
System	Number of servers	2-32 (10)
	Disk transfer rate	20 MBps
	Memory transfer rate	100 MBps
	HTTP protocol	1.1
	Intra-servers bandw.	100 Mbps
Workload	Arrival rate	100-5600 (700) clients per second
	Requests per session	Inverse Gaussian
	($\mu = 3.86$, $\lambda = 9.46$)	
	User think time	Pareto ($\alpha = 1.4$, $k = 2$)
	Embedded objects	Pareto
		($\alpha = 1.1 - 1.5(1.33)$, $k = 1$)
	Hit size - body	Lognormal ($\mu = 7.640$, $\sigma = 1.705$)
	Hit size - tail	Pareto ($\alpha = 1.383$, $k = 2924$)

Table 8.1: Parameters of the system and workload model.

the system sensitivity as a function of various parameters that is, client arrival rate, number of servers and disk cache size. To compare performance results, we maintain equivalent the ratio between the offered load and Web cluster capacity that is, for higher number of servers we augment proportionally the arrival rate.

Figure 8.1 shows the Unbalance Factor when the client arrival rate varies from 100 to 900 clients per second. For example, an arrival rate of 600 clients per second corresponds to an arrival rate of $340 - 370$ page requests per second. (It must be considered that clients do not send requests during user think time and Web service time.) From this figure we observe that all dispatching policies achieve analogous results. With a low arrival rate, the unbalance factor is negligible. With an arrival rate of 700-900 clients per second, the system is highly loaded. Hence, the queue lengths augment rapidly and the unbalance factor increases as well. However, this happens for all policies without significant differences. In all these instances, system scalability can be achieved only through an increment of the number of Web server nodes.

Varying the number of Web servers does not change the performance result order of the dispatching policies. Figure 8.2 shows that the unbalance factor increases with the number of servers, with a curve skew minor than 1. Even for higher numbers of servers the performance of a static information-less policy such as RR is quite comparable to both WRR state-aware algorithms. Other not reported experiments for different system parameters confirm these conclusions.

Our motivation for this unexpected result is that the heavy tail service time dis-

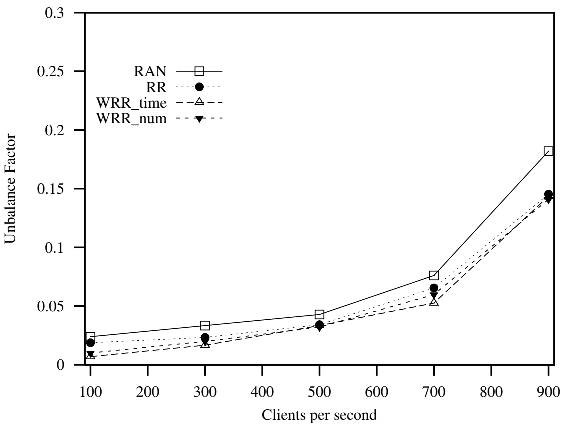


Figure 8.1: Sensitivity to the client arrivals per second.

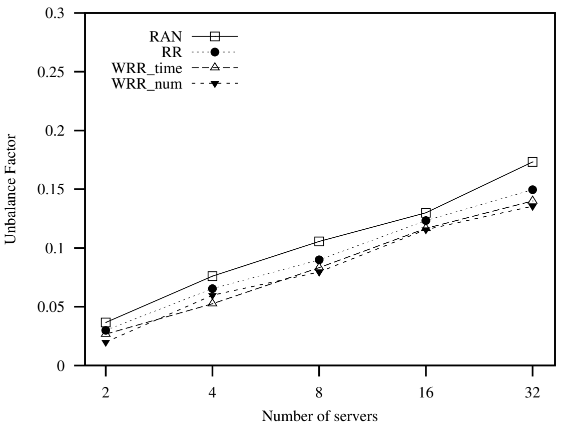


Figure 8.2: Sensitivity to the number of Web server nodes.

tributions and bursts of client arrivals are well counterbalanced by the Dispatcher full control on request arrivals. We carried out other sensitivity experiments as a function of the disk cache size. In Figure 8.3 we show the peak throughput of the Web cluster as a function of the document cache dimension, measured as a percentage of total size of site's documents set. We see that when the cache hit rate passes a low threshold (e.g., 10-15%), sophisticated mechanisms for controlling dynamic system states are really unnecessary. So a simple round robin dispatcher that distributes the load circularly among the server nodes is the best compromise between efficacy and cost for achieving scalable Web publishing sites with static content.

A global increase of the service time scale per request, and a mixed workload, should result in a major stress for the scheduling policy. Hence, setting a dynamic

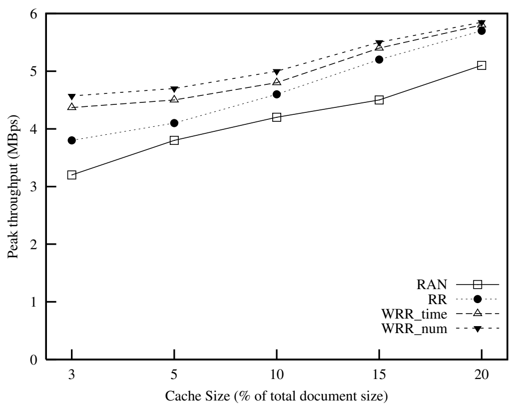


Figure 8.3: Sensitivity to the disk cache dimension.

dispatching policy could improve Web cluster performance. On the other hand, augmenting dynamism of the system environment makes much harder to tune the best parameters of the state-aware policies. To investigate how this trade-off affects scalability of Web clusters is the subject of the following section.

8.1.2 Dynamic contents scenario

Let us now consider Web clusters that provide mainly dynamic information. Now, the server response must be personalized for each user, and we suppose that each answer is obtained through more or less intensive computation and/or database operations.

Focusing on this class of Web systems has two main consequences: we remove the effects of Web caching because each information is useful to one client; we increase the service time scale of Web requests of two or more orders of magnitude with respect to the time required to serve a static request (few milliseconds), denoted Basic Service Time (BST). Starting from this basic measure of service, we consider the following time scales for dynamic Web services:

$10BST$: Typically for requests of long data streams, such as multimedia or software files.

$100BST$: Typically for requests that involve light-medium queries to a database.

$1000BST$: Typically for requests that involve some intensive computation and/or complex search in one or multiple databases.

This classification although realistic, does not want to be a precise taxonomy for all Web services as in [78]. Indeed, some requests for multimedia files could be classified as $100BST$, or some database queries as $10BST$.

In our experiments we consider that the Web cluster receives a mixed workload where requests could be static (BST), lightly dynamic ($10 - 100$ BST) and heavily dynamic ($100 - 1000$ BST). In particular, we consider two representative scenarios:

***Scenario A*:** it consists of 50% of static requests, 25% of lightly dynamic requests and 25% of heavily dynamic requests.

***Scenario B*:** it consists of 30% of static requests, 35% of lightly dynamic requests and 35% of heavily dynamic requests.

The first set of results aims to demonstrate the convenience of using a dynamic WRR policy over a static RR, if the parameters of the dynamic policies are tuned at their best. The main factors that affect WRR performance are the load indexes (mean response time required for the WRR_time policy, and mean number of active processes at each server for the WRR_num policy), and the interval of state information gathering by the Dispatcher that is, the T_{get} value defined in section 4.2.3.

Figure 8.4 and 8.5 show the sensitivity of WRR_num with respect to the T_{get} periods for scenario A and B. WRR_time is not reported because it achieves similar results.

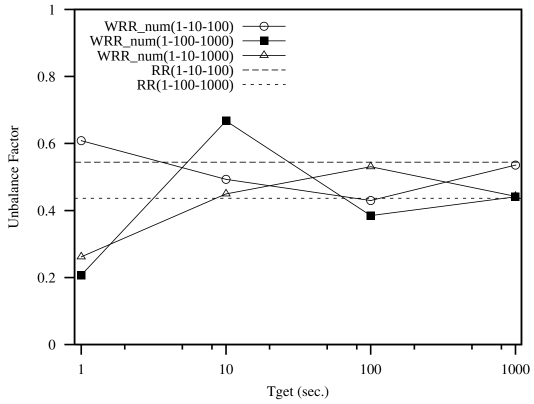


Figure 8.4: Sensitivity of WRR_num to the Tget parameter for the scenario A

The performance of WRR algorithms is very sensitive to T_{get}. The best values depend on the time scale of the service time and on workload composition. Figure 8.4

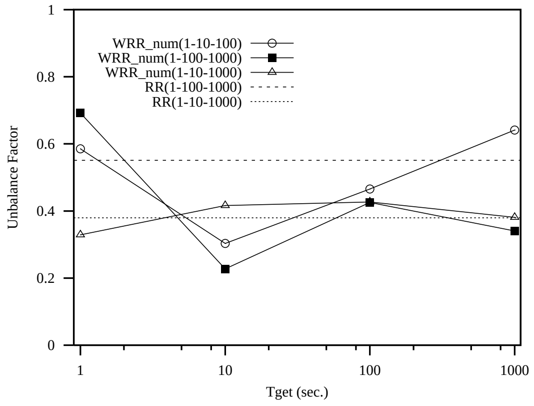


Figure 8.5: Sensitivity of WRR_num to the Tget parameter for the scenario B

and 8.5 show that a bad choice for the T_{get} value could result in a 50% decrease of performance (even worse than RR) when lightly dynamic requests are 100 BST and heavily dynamic requests are 1000 BST.

In the remaining part of this section, we suppose that we are able to choose always the best T_{get} value for the WRR policies. We will refer to these algorithms as WRR_time-Best and WRR_num-Best. In particular in the last set of experiments, we compare WRR-Best policies with RR and RAN under scenario A and B with three combinations of dynamic workload:

***1-10-100*:** that is 50(30)%, 25(35)%, 25(35)% of BST, 10BST, 100BST requests for scenario A(B);

***1-10-1000*:** that is 50(30)%, 25(35)%, 25(35)% of BST, 10BST, 1000BST requests for scenario A(B);

***1-100-1000*:** that is 50(30)%, 25(35)%, 25(35)% of BST, 100BST, 1000BST requests for scenario A(B).

Figures 8.6 and 8.7 compare performance RAN, RR and WRR-Best strategies when the system is stressed by the previously mentioned workload combination. The first observation is that performance of each policy depend much on workload. In scenario A (figure 8.6), when we pass from the workload 1-10-100 to the other two combinations, the unbalance factor decreases. This result is motivated by the intensive use of Web server resources. Both queue length and response time increase so that the mutual unbalance is less evident in percentage. Moreover, we can see that the mean number of active process is always a good index for server load.

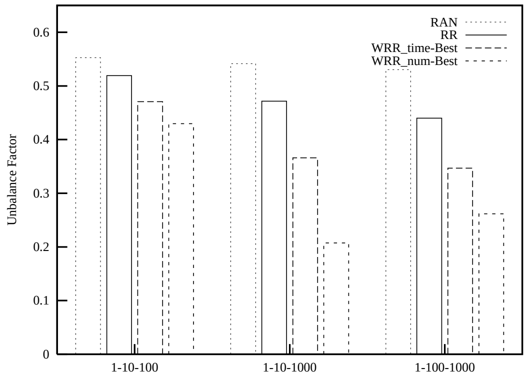


Figure 8.6: Static vs. Dynamic policies in scenario A for different combinations of workload.

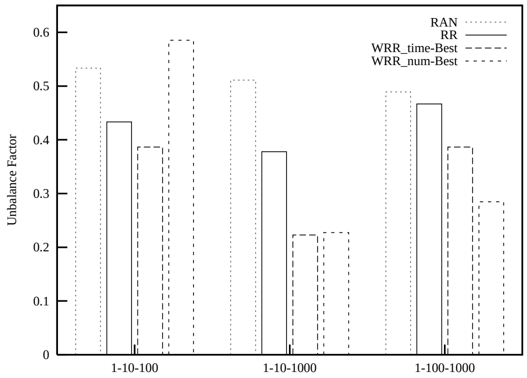


Figure 8.7: Static vs. Dynamic policies in scenario B for different combinations of workload.

In scenario B (figure 8.7), when there is a 70% percent of dynamic requests, it is impossible to define a stable trend for dispatching strategies behavior. For example, in the case 1-10-100 and 1-10-1000, WRR_time is the best policies. For the workload 1-100-1000, best performance are achieved by the WRR_num policy as in scenario A.

8.1.3 Significance of this performance study

Bursts of arrivals and heavy tailed service times are the main motivations that leaded many Web clusters to use dynamic state-aware policies. We wanted to investigate which workload characteristics really motivate the overhead for dynamic algorithms to achieve scalable Web clusters. We observed that for most Web publishing sites characterized by a large percentage of static information, a static dispatching policy such as RR guarantees a satisfactory scalability and load balancing. Our interpretation for this result is that a light-medium load (even with some high peaks and burst of arrivals) is implicitly balanced by a fully controlled circular assignment among the server nodes that is guaranteed by the Dispatcher of the Web cluster.

When the workload characteristics change significantly, so that very long services dominate, dynamic algorithms such as WRR should be applied to achieve a more uniform distribution of the workload and a scalable Web site. However, in highly variable Web sites, dynamic algorithms have serious tuning problems which are unknown to static policies. Hence, although necessary in most systems, dynamic policies guarantee scalability only if it possible to tune them as a function of the kind of Web services provided by the site.

8.2 Scalability analysis under realistic workload.

We now investigate the scalability of WRR in a realistic scenario. Indeed our intention is to use the workload model defined in Chapter 7, to model the typical load of a Web site that provides one or a mix combination of the following Web services.

Static Web services. Requests for HTML pages with some embedded objects. Typically, this load has a low impact on Web server components. Only requests for very large files are disk and network bound.

Dynamic Web services. Requests for HTML pages, where objects are dynamically generated through Web and back-end server interactions. Typically, these requests are CPU and/or disk bound.

Secure Web services. Requests for a dynamic page over a secure connection. Typically, these services are CPU bound because of overheads to setup a secure connection and to execute cryptography algorithms.

Tables 8.2, 8.3 and 8.4 show the parameters of the static, dynamic and secure workload respectively.

The workload models are mixed together to emulate three scenarios: *static Workload scenario* characterized by static workload only; *dynamic Workload scenario* scenario characterized by a mix of static (50%) and dynamic (50%) workload; *secure*

Category	*Distribution*	*Parameters*
Pages per session	Inverse Gaussian	$\mu = 3.86$, $\lambda = 9.46$
User think time	Pareto	$\alpha = 1.4$, $k = 1$
Objects per page	Pareto	$\alpha = 1.245$, $k = 2$
HTML object size	Lognormal Pareto	$\mu = 7.630$, $\sigma = 1.001$ $\alpha = 1$, $k = 10240$
Embedded object size	Lognormal	$\mu = 8.215$, $\sigma = 1.46$

Table 8.2: Static workload model.

Category	*Mean Service Time*	*Frequency*
Light Intensive	16 msec.	0.1
Middle Intensive	46 msec.	0.85
Intensive	150 msec.	0.05

Table 8.3: Dynamic workload model.

Workload scenario scenario characterized by a mix of static (50%) and secure (50%) workload. The secure workload consists of dynamic requests only.

8.2.1 Static workload scenario

Figure 8.8 compares the 90-percentile of page delay for different Web cluster configurations. We observe that when the system with 4 server nodes begins to be overloaded, corresponding to 190 clients per second (cps), if we scale to 8 or more Web servers, the 90-percentile of page delay decreases of one order of magnitude, from 11.6 seconds to 1.35 seconds.

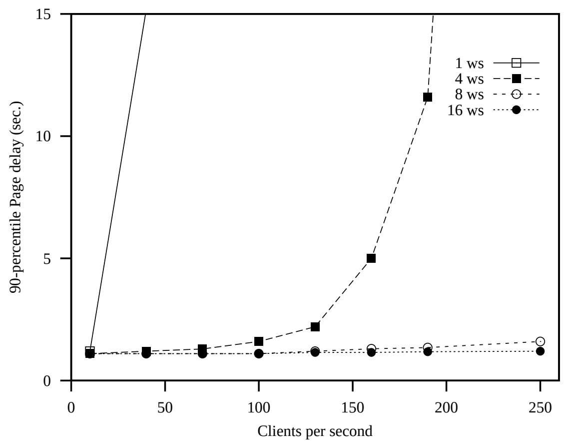


Figure 8.8: Static scenario: 90-percentile of page delay.

When a Web cluster with 4 nodes receives more than 160 clients per second, the

Category	*Throughput (Kbps)*
RSA(256 bit)	38.5
Triple DES	46886
MD5	331034

Table 8.4: Secure workload model.

system is overloaded. Figure 8.9 indicates that over that load threshold the peak system throughput decreases dramatically. The throughput for a Web cluster with 8 and 16 nodes continue to increase for higher numbers of client arrivals. However, the resulting throughput is much lower than the double of that related to the Web cluster with 4 nodes. The motivation is that the system with 4 nodes has a utilization much higher than that of the system with 8 and 16 nodes.

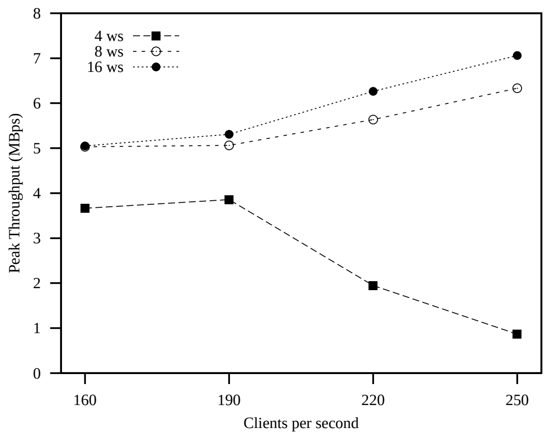


Figure 8.9: Static scenario: Peak system throughput.

However, the main goal of Figure 8.9 is to demonstrate that a Web cluster with four servers requires a T3 Internet connection. When we scale-out the system to more than a certain number of servers, the Web cluster requires a larger bandwidth connection or (better) a geographically distributed Web site. Otherwise, the risk is that performance are not acceptable because of the network latency. However, we will see that passing from a locally to a geographically distributed Web system causes a relatively high loss of performance. To this purpose, in Figure 8.10 we compare the 90-percentile of page delay at a Web cluster and at a geographically distributed Web cluster. Both architectures have the same number of server nodes and are subject to same static workload with an arrival of 400 clients per second.

To model a geographical distribution and different time zones, we divide the Internet into four world areas. Each area contains a Web cluster with four Web server nodes and various client domains. To represent the variability of traffic coming from different regions, we assign each client to one Internet region with a popularity probability that depends on the day hour in each time zone [37]. The popularity curve is taken from [12]. In the figures, we consider four consecutive hours starting from 24pm until 3am. Each region is supposed to be in a time zone shifted of 6 hours from the previous region. Due to different connection popularities, in the considered four hours we have the following probabilities of receiving requests from each of the four regions: hour 24pm, 0.26, 0.08, 0.26, 0.4; hour 1am, 0.18, 0.13, 0.26, 0.43; hour 2am, 0.1, 0.17, 0.28, 0.45; hour 3am, 0.06, 0.2, 0.31, 0.43. Figure 8.10 evidences the difficulties of geographically architectures: at any of the four hours, the page delay is much higher than that guaranteed by the Web cluster. One motivation for this result is that request dispatching among the Web clusters is based on network proximity only that is, clients requests are assigned to the closest Web cluster. Although this policy is implemented in most real systems (e.g., [48]), the consequence is that the distributed Web cluster is highly unbalanced when the sources of traffic requests from the four regions are more skewed, say hour 2am in our experiments. The result motivates the search for more sophisticated algorithms for geographically load balancing that area out of the scope of this thesis.

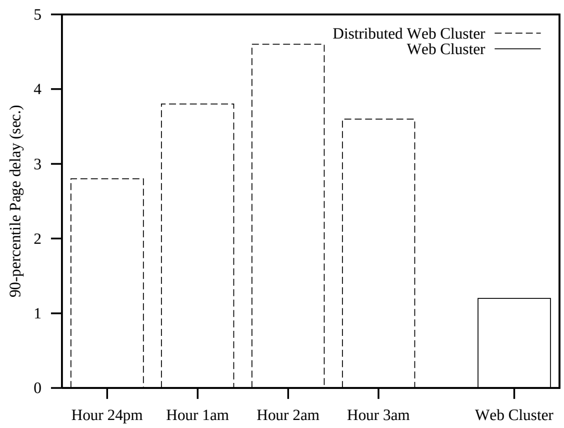


Figure 8.10: Web cluster vs. Distributed Web cluster.

8.2.2 Dynamic workload scenario

Dynamic requests are served through the cooperation of Web and back-end server nodes. Looking at Table 8.3, we can expect that in this scenario, the system bottleneck is at the back-end level. Nevertheless, in the first set of experiments we configure the Web cluster with the same number of Web server (ws) and back-end (be) nodes. The goal is to find the break-point for the system, thereby motivating an increase in the number of back-end nodes that avoids the system bottleneck.

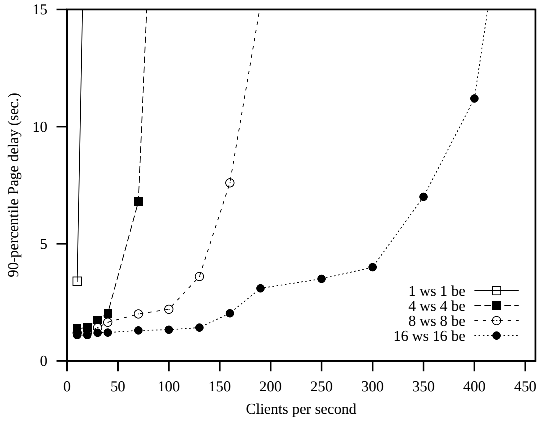


Figure 8.11: Dynamic scenario: 90-percentile of page delay.

Figure 8.11 shows that for a Web cluster with 8 servers at each level the break-point is 130 cps, when the 90-percentile of page delay is equal to 3.6 seconds. With 16 nodes at each level, we scale the break-point to 300 cps (4 seconds for page delay). This threshold is more than double than the previous limit. If we compare the results of static versus dynamic scenario, we may observe that, without an appropriate tuning of the system, performance decreases up to 50%. For example, in static scenario the break-point with 8 nodes is up to 250 cps, while in dynamic scenario the acceptable load halves that is, 130 cps. Figure 8.12 shows the peak throughput of the Web cluster. Analogously to the static scenario, a 16 node cluster needs a large bandwidth network or a geographically distributed architecture. For Web clusters with 4 and 8 nodes the dramatical crash of the system due to the over-utilization is evident.

The next step is to tune the number of back-end nodes to reduce the bottleneck. In Figure 8.13, we start from an overloaded system with a 90-percentile of page delay of about 100 seconds and increase the number of back-end nodes until an acceptable page delay is reached. The starting point is a cluster with a number of back-end nodes

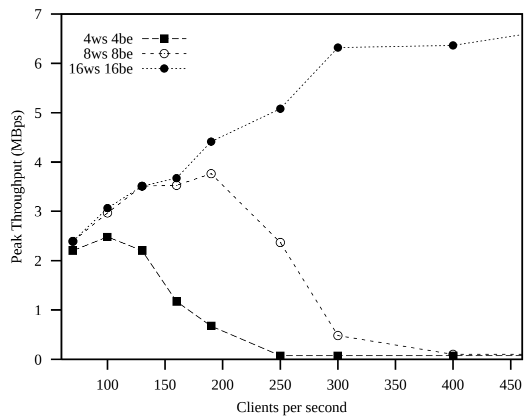


Figure 8.12: Dynamic scenario: Peak system throughput.

(num_be) equal to the number of Web server nodes (num_ws), then we increase the ratio between num_be and num_ws from 1.5 to 5. Figure 8.13 shows that a cluster composed by 4 Web servers and 10 back-end nodes can manage the high workload condition, while with 8 Web server nodes we need up to 20 back-end nodes. When the ratio is over 4 Web back-end servers for each Web server, the performance begins to deteriorate because the front-end nodes become the bottleneck.

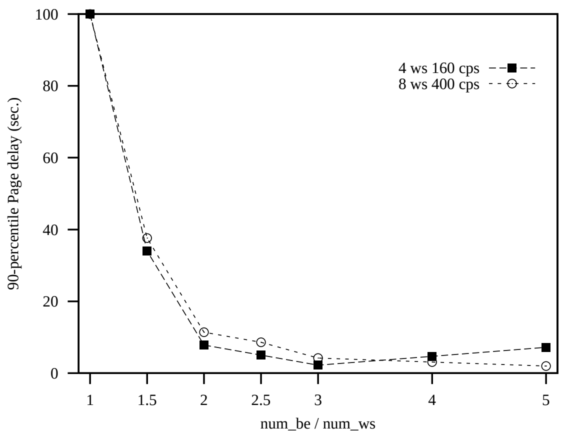


Figure 8.13: Dynamic scenario: 90-percentile of page delay for system tuning.

8.2.3 Secure workload scenario

We now evaluate the performance of a Web system, say an e-commerce site, that is subject to a mix of static, dynamic and secure workload. As for the dynamic scenario, we aim at discovering the break-point of the system. For the scenarios subject to half of secure requests, the bottleneck of the system is represented by the CPU of Web servers that must implement all secure connections and data encryption/decryption operations.

Figure 8.14 shows that the operations of encryption and decryption are very critical tasks. A very limited increase in client arrivals is sufficient to congestion CPU system's queues. The motivation for this critical system behavior is that each new secure session requires an authentication procedure through the computationally expensive RSA algorithm. As expected, the admitted arrival rate is about half of the load supported by a Web cluster subject to a dynamic scenario (in which half requests are not secure).

Although Figure 8.14 shows that at the break-point the 90-percentile of page delay is about 2 seconds, we have to consider that the setup of a new SSL session requires an exchange of 7 messages. Moreover, each successive object (if the session ID is still valid) requires an exchange of 5 messages. Hence, network delays have an impact on the page response time experimented by the client much higher than that corresponding to the previous two scenarios.

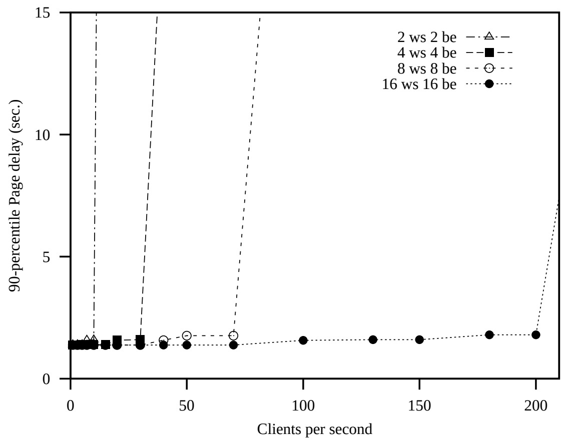


Figure 8.14: Secure scenario: 90-percentile of page delay.

Since for the secure scenario the bottleneck is represented by the Web server nodes, we can reduce the number of back-end nodes. To this purpose, we consider three architectures where the number of back-end nodes is a fraction of the Web server nodes number that is, 0.5, 0.75, and 1 ratios. Figure 8.15 shows that the best configuration is achieved for the 0.75 ratio, say 12 back-end and 16 Web server nodes. Below this ratio, the back-end nodes become the system bottleneck again.

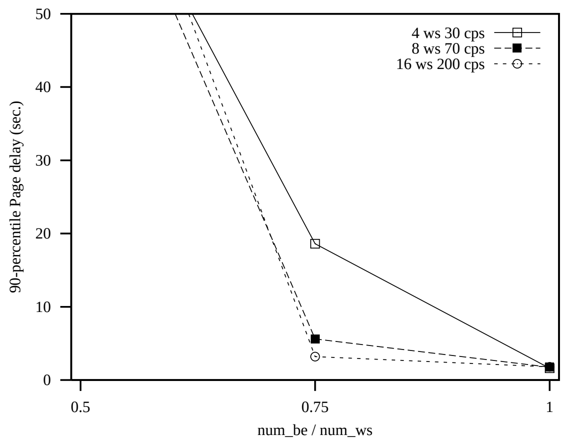


Figure 8.15: Secure scenario: 90-percentile of page delay for system tuning.

8.2.4 Significance of the performance study on scalability

From the performance study carried out in this section we can take the following main recommendations.

- To dimension and manage Web system architectures for *static scenarios* is not a big issue, even if Web sites have to serve very large files.

- When we pass to consider a *dynamic scenario*, the difficulty of choosing the right dimension of the system for a user acceptable response time (less than 8 seconds) augments. The main reason is that a dynamic request can have a service time of two order of magnitude higher than a static request with not negligible probability. Overprovisioning the system with respect to that required by the average load is reasonable if we want to guarantee performance to all classes of users. In this case, the main problem is to choose the right ratio between Web server and back-end nodes. As a rule of thumb, we can reason on the basis of average service times even if mean values are not always realistic when heavy-tailed distribution functions are involved. An empiric demonstration of this result is given by Figure 8.13 where we see that when the dynamic load represent four-five times the static load, we need at least two back-end servers for each Web server. At the other extreme, we have that the maximum number of Web servers per back-end is four. After this threshold, the Web server node becomes the system bottleneck.

- The *secure scenario* is the most severe test-bed. This was certainly expected, even if we were surprised to observe that even a very slight increment of the load could have crash consequences on the Web site (see Figure 8.14). For this reason, we conclude that Web sites that provide secure services are the only systems for which overprovisioning is highly reasonable in order to guarantee performance.

- When we consider systems with a large number of server nodes, the network connection risks to be the system bottleneck, so we have to consider a geographically distributed Web system. Passing from a *locally* to a *geographically* distributed Web system, we have to take into account the performance loss of these latter architectures. We have seen that with present policies for geographic load balancing this loss can be extremely high. For example, if we have N server nodes in a Web cluster, we can even require $M = 3N$ servers geographically distributed to guarantee analogous performances. We feel that this result can be improved by using more sophisticated algorithms for geographically load balancing. However, it seems difficult to reach M/N ratios below 1.5.

Chapter 9

Content-aware dispatching policies: A performance study

In this Chapter we propose a performance study of content-aware policies, first through simulation and later through experiments on the ClubWeb prototype. In the simulation analysis we refer to the simulation and workload model proposed in Chapter 5 and Chapter 7 respectively.
Figure 5.2 shows the system model that we consider.

The dispatching policies consider in Chapter8 take static decisions independently of any state information (e.g., RAN and RR) or they take dynamic decisions on the basis of the state of the server nodes (e.g., WRR). We now focus on content-aware dispatching policies that can be applied to a layer-7 Web switch4.3

9.1 CAP: A new class of client-aware policies

As a first step, we detail the characteristics of a new class of content aware dispatching policies, called **CAP**, from *client-aware policy*.

CAP in its initial form is a pure client info aware policy, however it can be easily combined with some server state information. The key idea comes from the observation that policies such as WRR and LARD work fine in Web clusters that host traditional Web publishing services. Indeed, most load balancing problems occur when the Web site hosts heterogeneous services that make an intensive use of different Web server's components. For describing the CAP by example, let us classify the services of a Web site into four main categories.

Web publishing sites providing static information (e.g., HTML pages with some embedded objects) and dynamic services (e.g., result or product display requests) that do not intensively use server resources. The content of dynamic requests is not know at the instant of a request, however it is generated from database queries whose arguments are known before hand.

Web transaction sites providing dynamic content generated from (possibly complex) database queries built from user data provided by an HTML form. This is a disk bound service because it makes intensive use of disk resources.

Web commerce sites providing static, dynamic and secure information. For security reasons, some dynamically generated content may need of a secure channel that in most cases is provided by the SSL protocol. Cryptography makes intensive use of CPU resources. Hence, Web commerce services are disk and/or CPU bound.

Web multimedia sites providing streaming audio and video services. In this thesis, we do not consider this type of applications that often are implemented through specialized servers and network connections.

Although realistic, this classification is done for the purposes of our thesis only and does not want to be a precise taxonomy for all Web services. The idea behind the class of CAP policies is that, although the Web switch can not estimate precisely the service time of a client request, from the URL it can distinguish the class of the request and its impact on main Web server resources. Any Web content provider can easily tune the CAP policy at its best. Starting from the above classification, we give an example of a CAP algorithm when the Web requests can be distinguished into four classes: *static* and *lightly dynamic* Web publishing services (N); *disk bound* services (DB) for example in Web transaction and Web commerce sites; *CPU bound* (CB) and *disk and CPU bound* (DCB) services for example in Web commerce sites.

In the considered version of CAP, the Web switch manages a circular list of assignments for each class of Web services. The goal is to share multiple load classes among all servers so that no single component of a node is overloaded.

We suppose that the server A has already received one request of type CB and one of type DCB; the server B has received one request of type N, and two of type DB. The sequence of successive requests to the Web cluster is shown in Figure 9.1. By using the CAP assignment, server A and B have a similar number of requests for each class of service, while this does not happen when using RR or LARD. For

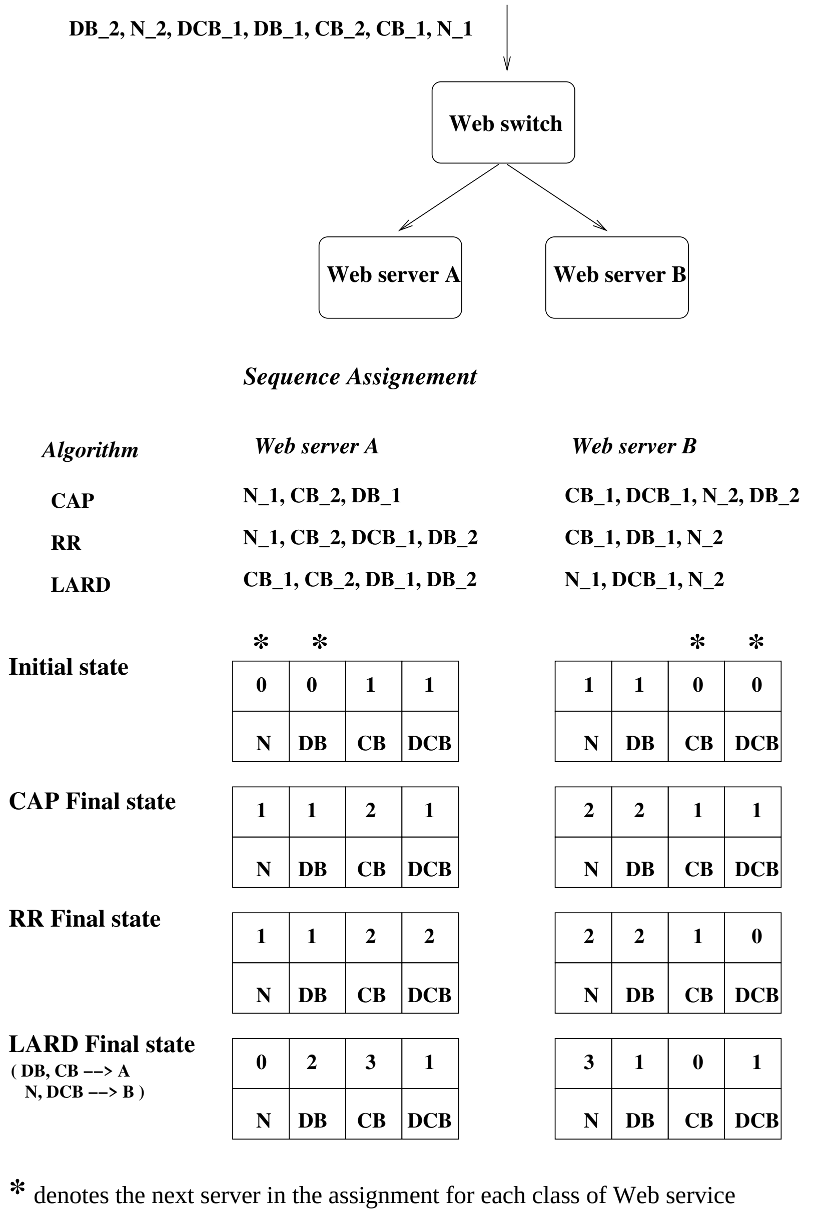


Figure 9.1: Behavior of CAP, RR and LARD dispatching policies.

example, in the case of RR the server A receives four intensive requests that stress the CPU and/or disk, while server B receive only one CPU bound request. In the case of LARD, we suppose that the requests of type DB and CB are assigned to the server A and those of other types to the server B. This dispatching results that the

Category	*Parameter*	*Value*
Web cluster	Number of servers	2-32
	Disk transfer rate	20 MBps
	Memory transfer rate	100 MBps
	HTTP protocol	1.1
	Intra-servers bandwidth	100 Mbps
Client	Arrival rate	100-1600 clients per second
	Web requests per session	Inverse Gaussian ($\mu = 3.86$, $\lambda = 9.46$)
	User think time	Pareto ($\alpha = 1.4$, $k = 2$)
	Embedded objects per Web page	Pareto ($\alpha = 1.1 - 1.5(1.33)$, $k = 1$)
	Hit size request (body)	Lognormal ($\mu = 7.640$, $\sigma = 1.705$)
	(tail)	Pareto ($\alpha = 1.383$, $k = 2924$)

Table 9.1: Parameters of the system and workload model.

server A receives two CPU bound and two disk bound requests, while the server B receives only one request of type DCB.

CAP does not require a hard tuning of parameters which is typical of most dynamic policies because the service classes are decided in advance, and the dispatching choice is determined statically once the URL has been classified.

9.2 Simulation experiments

We model the typical load of a Web site that provides one or a mix combination of the static dynamic and secure Web services using the workload model defined in Chapter 7.

Table 9.1 shows the parameters of the system model and static load, and table 9.2 shows the parameters of the secure workload. For the dynamic workload we use the parameters of the table 8.3.

Category	*Throughput (Kbps)*
RSA(256 bit)	38.5
Triple DES	46886
MD5	331034

Table 9.2: Secure workload model.

We compare performance of different scheduling policies for Web clusters under three main classes of workload:

Web publishing site containing *static* and *lightly dynamic* documents. A static document resides on the disk of the Web server, it is not modified in a relatively long time interval and is always cacheable. The cache of each node is set to 15% of the total size of the Web site document tree. A lightly dynamic document is cacheable with 0.3 probability.

Web transaction sites contain 60% of static documents and 40% of dynamically created documents. The database query to back-end servers requires intensive disk use and its result is not cacheable.

Web commerce sites have 30% of static requests 30% of lightly dynamic requests and various combinations for the remaining 40% of requests.

9.2.1 Simulation results

As the main measure for analyzing the performance of the Web cluster we use the cumulative frequency of the *page latency time*.

The CAP policy, in its simplest form, does not require to set any parameter other than the choice of classes of services that can be identified from the URL. The LARD strategy requires to set only the parameters T_{low} and T_{high} for server utilization as described in Section 4.3. In our experiments we set T_{low}=0.3 and T_{high}=0.7, and we did not observed considerable changes for lightly different values, such as 0.2 and 0.75. As seen in Section 8.1.2 the most complex policy to tune is the WRR that is sensible to the adopted load metric (number of active connections, mean page latency time at each server) and to the selected T_{gat} value for gathering server state information.

9.2.2 Web publishing

We assume two scenarios where first all requests are for static documents and then for lightly dynamic documents.

Figure 9.2(a) shows that the LARD strategy that exploits the reference locality of Web requests performs better than CAP and WRR. LARD guarantees with very high probability that the page latency time is less than 1 second. The analogous probability is about 0.9 for CAP and 0.8 for WRR.

In the next experiments we consider only requests for lightly dynamic documents, that have a lower probability of being found in the disk cache. Figure 9.2(b) shows that CAP and LARD have a similar behavior even if LARD still performs slightly better. However, if we compare this Figure with Figure 9.2(a) we can observe that, while CAP and WRR maintain similar results, actually LARD is truly penalized by a lower cache hit rate.

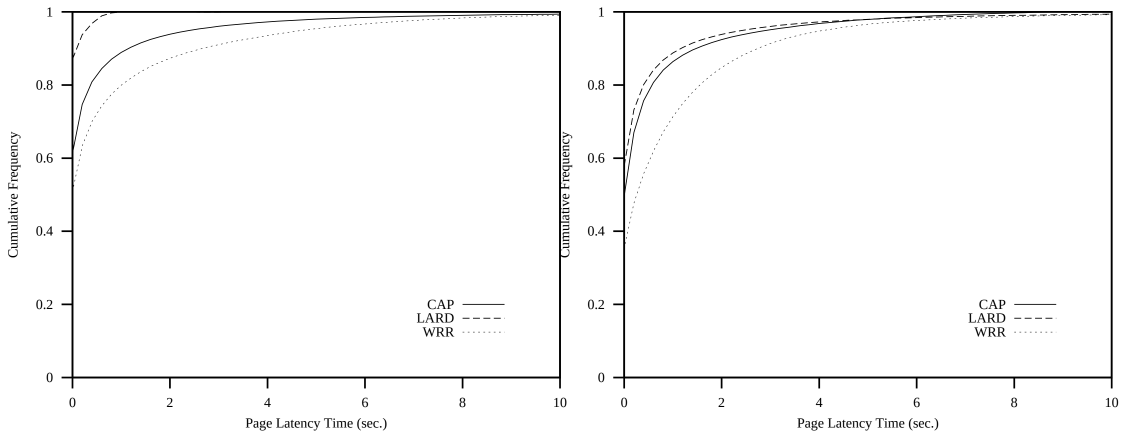


Figure 9.2: Web publishing: cumulative frequency of page latency time.

9.2.3 Web transaction

When the document hit rate is low and a relatively high percentage of requests (40%) is disk bound, the CAP policy starts to exploit its benefits for multiple service characteristics. Figure 9.3 shows that the page latency time of CAP is much better than that achieved by WRR and LARD strategies. CAP policy improves Web cluster performance of 35% over WRR and 55% over LARD. Indeed, the 90-percentile of page latency time using CAP is less than 2 seconds, while it is about 7 seconds for WRR and about 20 seconds for LARD. LARD has the worst performance because incoming requests are assigned to a Web server first on the basis of document locality and then on the basis of server load state. Our results show that routing requests to the same server until it reaches a highly loaded state (e.g., T_{high}=0.7) does not work in Web transaction sites. On the other hand, WRR is quite robust, because its performance does not degrade too much for sites with heterogeneous services.

9.2.4 Web commerce

This is the most heterogeneous workload. We consider three scenarios where lightly dynamic, CPU bound and disk bound requests are mixed in different ways.

In the first set of experiments we assume that 40% of total requests need secure

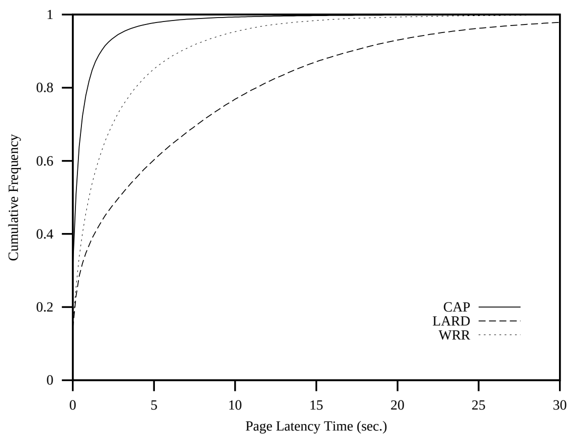


Figure 9.3: Web transaction: cumulative frequency of page latency time

connections and cryptography. This is the most critical scenario because CPU bound requests affect performance of the other 60% of requests that have to use the CPU, even if for less intensive operations such as parsing a request and building a HTTP response. Figure 9.4(a) shows how this very critical (and, we can say, unrealistic) scenario deteriorates performance of all dispatching strategies. However, multiple service scheduling allows CAP to achieve better results than WRR and LARD.

In the second scenario, we introduce both CPU and disk bound requests, but CPU bound requests (20%) differ from disk bound requests (20%). In Figure 9.4(b) we see that the CAP policy provides really good page latency times, while WRR is at its limit and LARD does not guarantee a scalable Web cluster. The improvement of the CAP strategy is considerable, especially if we consider that the WRR curve refers to the WRR policy with best parameters. For example, CAP achieves a page latency time of about 4 seconds with 0.95 probability. For WRR, the analogous latency time is achieved with a probability of about 0.78, and for LARD with a probability of less than 0.5.

In the last set of experiments, we consider a realistic Web commerce site where the requests can be for secure (20%), disk bound (10%) or both (10%) services. Even if the workload becomes more onerous, Figure 9.4(c) shows that the CAP policy guarantees a scalable Web cluster. Indeed, it achieves a 90-percentile for the page latency time of less than 5 seconds. The analogous percentile is equal to 20 seconds for WRR and it is higher than 35 seconds for LARD. For example, WRR and LARD have a page latency time of 5 seconds with a probability of 0.66 and 0.37, respectively. This is about two third and one third of the corresponding CAP probability.

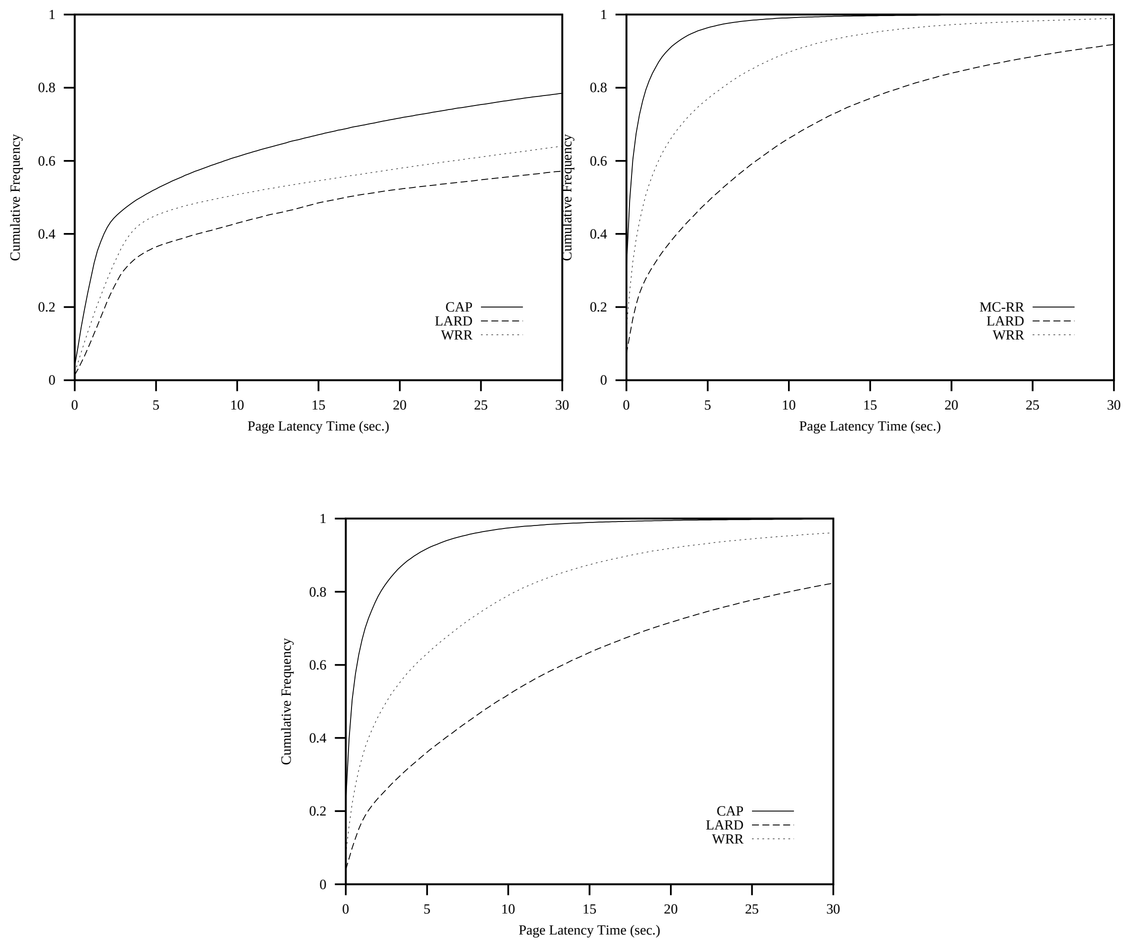


Figure 9.4: Web commerce scenario: cumulative frequency of page latency time.

9.3 Prototype experiments

The simulation results convinced us about the opportunity to test CAP on the Club-Web prototype, described in Chapter 6. In this section we give some performance results that confirm main conclusions obtained by the simulation model.

To test the system we use a pool of client nodes that are interconnected to the Web cluster through a dedicated Fast Ethernet. The clients runs on four PCs: two PentiumII 2x450Mhz with 256MB of memory and two PentiumII 233Mhz with 128MB of memory.

As synthetic workload generator we our a modified version of the WebStone benchmark introduced in Chapter 7. The main modifications concern the client behavior that in the original version of WebStone is not much realistic. In particular, we introduce the concept of user think-time and embedded objects per Web page as described in Section 7. Moreover, the file size of each object is modeled through a Pareto distribution. To emulate a dynamic workload, we create three CGI services: two of them stress the CPU of the Web server nodes in a lightly or intensive way, the third stresses both the CPU and the disk. In the experiments we compare the performance of dispatching strategies under three different scenarios: pure static workload, light dynamic workload, and intensive dynamic workload. In the dynamic scenario, 80% of requests are for static objects and 20% are for dynamic services. The dynamic requests are classified as CPU-light (10%), CPU-intensive (6%) and CPU-Disk-intensive (4%).

The main performance metrics are the Web cluster *throughput* measured in connections per second (conn/sec) and the *mean hit latency time*. We prefer to measure the throughput in number of connections instead of Mbps because, especially for a dynamic scenario, this metrics is considered more representative than the number of bit transferred during a certain interval. Indeed, a CPU intensive request can stress the system much more than the transfer of a file that could even be get from the disk cache.

At the beginning we measure the capacity for a single Web server that is, the maximum number of contemporary clients that could initiate a Web session without causing server overloading or crash. Under the dynamic light workload scenario, we obtain that a single server may handle about 60-80 clients with a mean system utilization equal to 60-70%.

In the first set of experiments we compare the performance of CAP and LARD under static and light dynamic workload scenario. Figure 9.5 shows the system throughput in the case of static scenario. We can see that LARD and CAP perform similarly. When the clients in the system are more than 240 the Web servers start to be highly loaded. This is confirmed by Figure 9.6 showing that the hit latency time increases rapidly in correspondence of this threshold value.

When we pass to consider a light dynamic workload, the performance results of dispatching policies do not change much. Figure 9.7 and 9.8 show the system throughput and mean hit latency time when the workload is lightly dynamic. Although the difference between CAP and LARD increases if compared to the static workload, the two policies are still comparable.

Figure 9.9 presents the system throughput for an intensive dynamic workload.

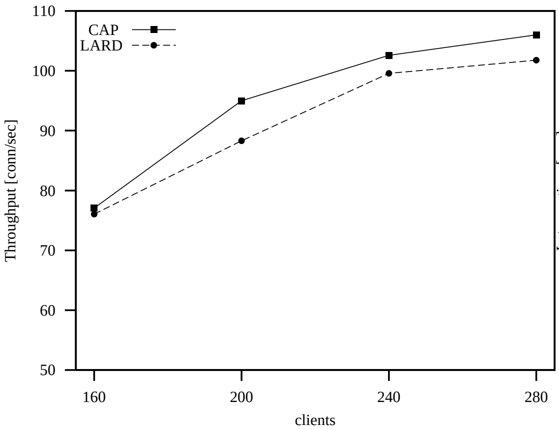


Figure 9.5: Static workload: system throughput.

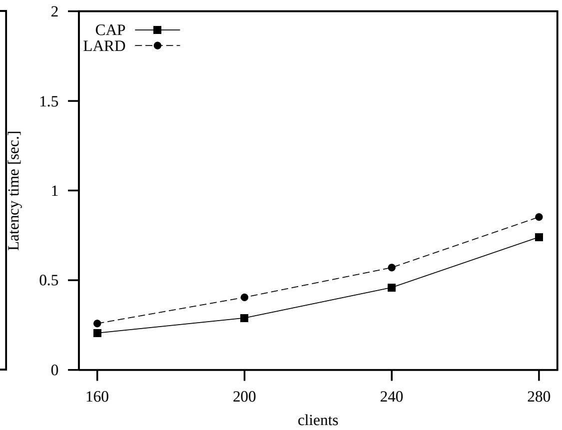


Figure 9.6: Static workload: mean hit latency time.

Indeed, the introduction of dynamic requests stresses the server resources in a quite different way. In this instance, CAP outperforms LARD performance. Compared to this latter algorithm, CAP throughput increases from 13% when 160 clients are in the system, to 21% when 240 clients are connected. In terms of hit latency time, Figure 9.10 shows that the CAP policy reduces to a half the hit latency time achieved by the LARD policy.

All the above results let us think that if the complexity and variety of client requests augment, the improvement of CAP can increase as well.

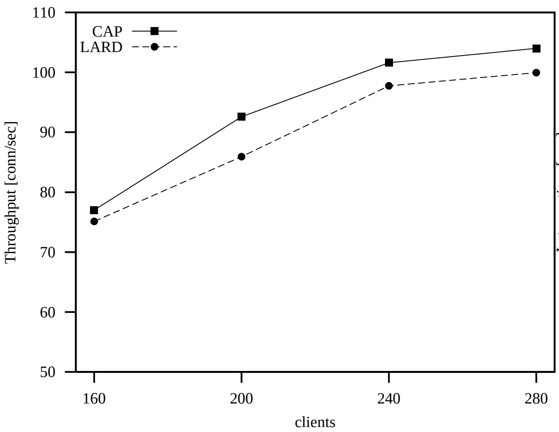


Figure 9.7: *Light* dynamic workload: system throughput

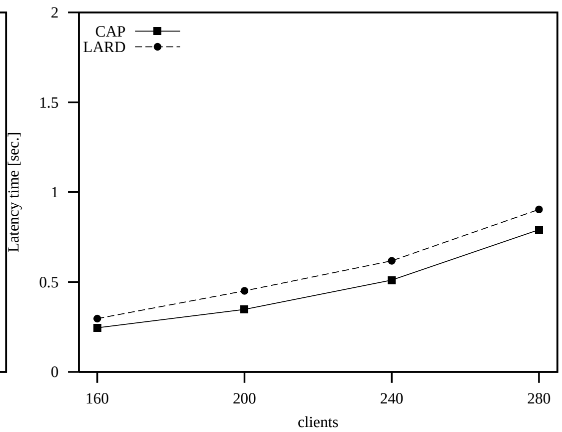


Figure 9.8: *Light* dynamic workload: mean hit latency time

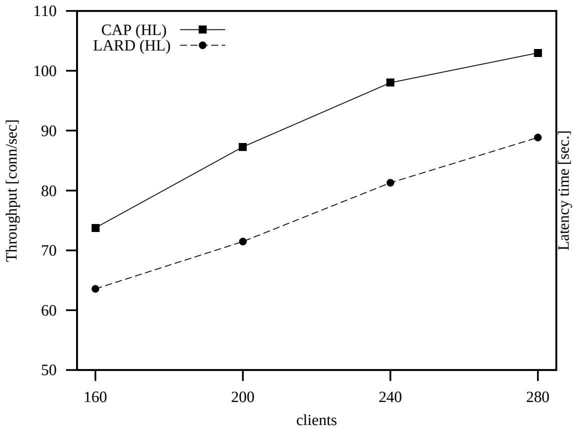


Figure 9.9: *Intensive* dynamic workload: system throughput

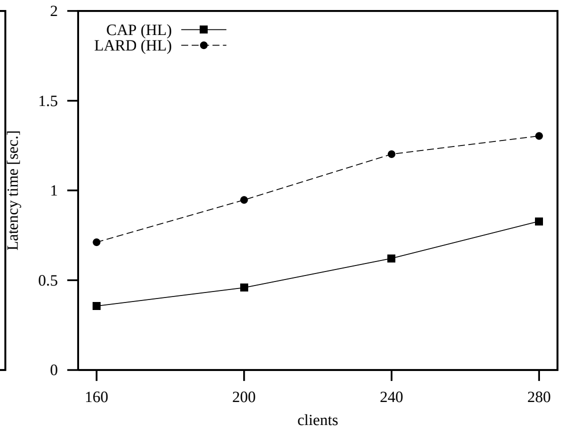


Figure 9.10: *Intensive* dynamic workload: mean hit latency time

The simulations and experiments on the ClubWeb prototype show that dispatching policies that improve Web caches hit rates give best results for traditional Web publishing sites providing most static information and some lightly dynamic requests. On the other hand, CAP provides better performance for modern Web clusters providing static, dynamic and secure services. Moreover, the pure CAP has the additional benefit of guaranteeing robust results for very different classes of Web services because it does not require a hard tuning of system parameters as the other server-aware dispatching policies do.

Chapter 10

Multi-Tier Architectures

The Web technology is becoming the preferred standard interface for accessing many services exploited through the Internet. While the first generation of Web sites was largely based on static and read-only information, an increasing percentage of Web sites provide information and services that are personalized for the client or created dynamically by the execution of some application process. The consequence is that dynamic pages and services are becoming essential in modern sites where Web-based technologies have emerged as a valid alternative to traditional client-server computing. Because of the complexity of the Web infrastructure, performance problems may arise in many points during an interaction with Web-based systems.

As show in the previous Chapters of this thesis, cluster-based Web systems are a robust and scalable solution to address Web site performance problems, even in complex Web services scenario.

In this Chapter we consider multi-tier cluster-based Web system (briefly, *Web cluster*), shown in figure 3.6, and we focus our attention on the issue of distributing incoming request at the second tier of the system.

Request dispatching and load sharing at the internal layers are implemented in most commercial products, but they are not widely studied topics in the research community. Indeed, the selection of a back-end server can be done by some *centralized* entity (the Web switch itself or some master node, e.g. [116]) or in a *distributed* way by any Web server. The combination of feasible alternatives is wide. This paper gives one of the first contribution to this area. We compare the performance of several combinations of centralized and distributed dispatching algorithms working at the first and second layer, and using different levels of state information. The main focus is on the second dispatching-level because the non-uniformity of the load and non-cachability of most documents (in spite of many efforts [30, 56, 94]) introduce additional degrees of complexity to the request dispatching issue. Indeed, burst arrivals and hot spots can be faced by the Web switch and servers through caching techniques so that a Web server node can typically deliver several hundreds of static files per second. On the other hand, to share the load for dynamic requests is quite difficult because they often require orders of magnitude higher service times. We also analyze the efficiency and the limitations of the different solutions and the tradeoff among the alternatives with the aim of identifying the characteristics of centralized and distributed approaches and their impact on performance. A partially unexpected result is that the most simple dispatching policies work better than more sophisticated ones, even when the system is highly loaded. Instead, detailed information about the server, especially past load, does not appear useful, and can often lead to bad dispatching choices. Our experimental results confirm in a quite different context the well known results by Eager et al. [61] and others related [51] obtained through analytical and simulative models.

Zhu et al. [116] have studied the problem of request dispatching in a multi-tier architecture, where a second-layer switch is integrated in each Web server node, called *master node*. When a Web server node receives a request for dynamic content, it queries the master to choose the back-end server to which forwarding the request. The back-end node selection is based on a prediction model that estimates the expected cost for processing the dynamic request on each slave node. Similar multi-tier architectures have been also analyzed in [27, 66].

The generation of dynamic content opens other performance issues that are beyond the scope of this paper. Indeed, the alternative solutions depend also on the application software, the chosen middleware and database technology. For example, commercial Web service software, such as BEA WebLogic and IBM WebSphere, have evolved from simple Web servers into complex Web application servers that use CGI, Java Server Pages, Microsoft Active Server Pages, XML, and other technologies. A different solution for load balancing based on CORBA middleware technology is proposed in [95] where several dispatching strategies have been also proposed and evaluated. Other interesting research fields in multi-tier Web architectures are not oriented to load balancing, but to mechanisms that can cache query results and dynamic content [30, 56, 94].

10.1 Dispatching algorithms for multi-tier Web cluster

In a multi-tier Web cluster there is a dispatching policy carried out by the Web switch that selects a Web server, and another dispatching choice when a back-end server has to be chosen for a dynamic request. The common rule is that these dispatchers cannot use highly sophisticated algorithms because they have to take immediate decisions for hundreds or thousands of requests per second. There are several alternatives for dispatching algorithms described in literature, however for the scope of this paper the most important choices are among *centralized* vs. *distributed*, and *state-blind* vs. *state-aware* algorithms.

The front-end architecture with a single Web switch that receives all inbound packets drives the choice to centralized dispatching policies. Hence, the real alternative for the Web switch dispatching-layer is among state-blind vs. state-aware algorithms. On the other hand, for the second dispatching-layer, we can consider both distributed policies carried out by the Web servers and centralized solutions where the decision is taken by the Web switch or by another server selected as a master.

State-blind (or static) algorithms are the fastest solution because they do not rely on the current state of the system at the time of decision making. However, the common belief is that these algorithms may make poor assignment decisions. *State-aware* (or dynamic) algorithms have the potential to outperform static algorithms by using some state information to help dispatching decisions. The main problems with dynamic algorithms are that they require mechanisms that collect and analyze data, thereby incurring in potentially expensive overheads and stale information about state conditions.

State-aware algorithms can take into account a variety of system state information that depends also on the protocol stack layer at which the dispatcher operates. They can be further classified according to the level of system state information being used by the dispatcher. A Web server operates at application layer and can apply *content-aware dispatching* algorithms, while a Web switch can operate at the same layer or at the TCP layer where only *content-blind dispatching* algorithms are allowed. We consider the following four classes of algorithms.

In *state-blind* policies the dispatcher assigns requests using a static algorithm and no dynamic information.

In *client state-aware* policies the dispatcher routes requests on the basis of some client information. For example, a dispatcher working at application layer can ex-

Policy	**Location**	**Knowledge**
RR	switch/server	state-blind
LL-CONN	server	server state-aware
LL-CPU	server	server state-aware
LL-DISK	server	server state-aware
WRR	server	server state-aware
PART	switch	client info-aware
CAP	server	client info-aware
CAP-CENTR	switch	client info-aware
LARD	switch	client and server state-aware
CAP-ALARM	server	client and server state-aware

Table 10.1: Dispatching policies to distribute client requests to Web and back-end servers.

amine the entire HTTP request and take decisions on the basis of more detailed information about the client. (Indeed *client info-aware* would be a more correct definition for this class of algorithms, but we prefer to use client state-aware for simmetry reasons.)

In *server state-aware* policies the dispatcher assigns requests on the basis of some server state information, such as current and past load condition, latency time, and availability.

In *client and server state-aware* policies the dispatcher routes requests by combining client and server state information.

We consider the dispatching algorithms which are representative of one of these four classes. Table 10.1 summarizes the policies analyzed in this paper and whether they are applied at the Web switch or at the Web server.

10.1.1 State-blind algorithms

State-blind policies do not consider any system state information. Typical examples are **Random** and **Round-Robin** (**RR**) algorithms. Random distributes the incoming requests uniformly through the server nodes with equal probability of reaching any server. RR uses a circular list and a pointer to the last selected server to make dispatching decisions, that is, if S_i was the last chosen node, the new request is assigned to S_j, where $j = (i+1) \, mod \, N$ and N is the number of server nodes. Therefore, RR utilizes only information on last assignment decision.

Both Random and RR policies can be easily extended to treat servers with different processing capacities by making the assignment probabilistic on the basis of server capacity. Different processing capacities can be also treated by using the so-called **static Weighted Round-Robin**, which comes as a variation of the Round-Robin policy. Each server is assigned an integer weight w_i that indicates its capacity. Specifically, $w_i = C_i/min(C)$, where $min(C)$ is the minimum server capacity. The dispatching sequence will be generated according to the server weights.

10.1.2 Server state-aware algorithms

When we consider dispatching algorithms that use some server state information we have to decide the *server load index*, how and when to compute it, how and when to transmit this information to the dispatcher. These are well known problems in a networked system [54, 64].

Once a server load index is selected, the dispatcher can apply different algorithms. A common scheme is to have the new request assigned to the server with the lowest load index. The **Least Loaded** algorithm (**LL**) actually denotes a set of policies that depend on the index chosen to characterize the server load. The three main factors that affect the latency time of a Web request are loads on CPU, disk, and network resources of the Web server nodes. Typical server state information includes the CPU utilization evaluated over a short interval, the instantaneous CPU queue length periodically observed, the disk or I/O storage utilization, the instantaneous number of active connections, the number of active processes, and the object latency time, that is, the completion time of an object request at the Web cluster side. These indexes need a process monitor on the servers and a communication mechanism from the servers to the dispatcher. For example, in the **Least Connections** policy (**LL-CONN**), which is usually adopted in commercial products (e.g., Cisco's LocalDirector [47], F5 Networks' BIG-IP [63]) the dispatcher assigns the new request to the server with the fewest active connections. In this paper we consider also other load metrics, such as CPU and disk utilization. In the **LL-CPU** policy and **LL-DISK** policy the dispatcher assigns requests to the server having lowest CPU and disk utilization, respectively.

The **dynamic Weighted Round-Robin** algorithm (WRR) is a variation of the version that considers static information (server capacity) as a weight. This policy associates each server with a dynamically evaluated weight that is proportional to the server load state, typically estimated as a number of connections [73]. Periodically, the dispatcher gathers load index information from the servers and computes the weights, that are dynamically incremented for each new connection assignment.

10.1.3 Client state-aware algorithms

The complexity and overheads of a dispatcher that can examine the HTTP request at application layer motivates the use of more sophisticated content-aware distribution policies. The proposed algorithms may use information about the requested URL for different purposes, such as: to improve reference locality in the server caches so to reduce disk accesses (*cache affinity*); to use specialized server nodes to provide different Web services (*specialized servers*), such as streaming content, dynamic content; to partition the Web content among the servers, for increasing secondary storage scalability (*service partitioning*); to increase load sharing among the server nodes (*load sharing*).

In cache affinity policies, the file space is typically partitioned among the server nodes. A hash function applied to the URL (or to a substring of the URL) can be used to perform a static partitioning of the files. The dispatching policy running on the Web switch (namely, **URL hashing** algorithm) uses the same function. However, the solution combining Web object partitioning and URL hashing can be applied to Web sites providing static content only. Moreover, it ignores load sharing completely, as it is difficult to partition the file space in such a way that requests are balanced out. Indeed, if a small set of files accounts for a large fraction of requests (a well-known characteristic of Web workloads, e.g., [13, 53]), the server nodes serving those critical files will be more loaded than others. These critical load peaks can be lowered by architectures that integrate proxy server solutions, but the other problem remains.

For Web sites providing heterogeneous Web services, the requested URL can be used to statically partition the servers according to the file space and service type they handle. The goal is to exploit the locality of references in the server nodes, to achieve the best cache hit rate for static documents, and possibly to employ specialized servers for certain types of requests, such as dynamic content, multimedia files, streaming video [113]. We refer to this policy as to **Service Partitioning** (**PART**). Most commercial content-aware switches deploy this type of approach (for example, F5 Networks' BIG-IP [63], Resonate's Central Dispatch [102]) although they use different names.

Another goal of the content-aware dispatching algorithms is to improve load sharing among the server nodes. These strategies do not require static partitioning of the file space and the Web services. Let us consider one policy belonging to this class, called **Client Aware Policy** (**CAP**) [39], that is oriented to Web sites providing services with different computational impact on system resources. The basic observation of CAP is that when the Web site provides heterogeneous services, each client request could stress a different Web system resource, e.g., CPU, disk, network. 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 impact on main Web system resources. A feasible classification for CAP is to consider disk bound, CPU bound, and network bound services, but other choices are possible depending on the Web content. To improve load sharing in Web clusters that provide multiple services, the Web switch manages a circular list of server assignments for each class of Web services. The CAP goal is to share multiple load classes among all servers so that no component of a node is overloaded. CAP in its initial form is a client state-aware policy, however it can be easily combined with some server state information.

It is worth to observe that the client state-aware policies do not require a hard tuning of parameters, which is typical of the server state-aware policies, because the service classes are decided in advance and the dispatching choice is determined statically once the requested URL has been classified.

10.1.4 Client and server state-aware algorithms

Client information is often combined with some server state information, for example to provide the so called *client affinity* [73, 82]. In policies based on client affinity, client and server information have a different weight: when available, client information usually overrides server information for assignment decisions. Instead of assigning each new connection to a server on the basis of its server state regardless of any past assignment, performance or functional reasons motivate the assignment of the same request to the same server.

The **Locality-Aware Request Distribution** (**LARD**) policy is a content-aware request distribution that considers both locality and load balancing [15, 97]. The basic principle of LARD is to direct all requests for the same Web object to the same server node as long as its utilization is below a given threshold. By so doing, the requested object is more likely to be found into the disk cache of the server node. Some check on the server utilization is useful to avoid overloading servers and, indirectly, to improve load sharing.

A modified version of the CAP policy, namely **CAP-ALARM**, uses some server load information in addition to the request content for dispatching. The requests are distributed using the CAP algorithm unless the chosen server has a load close to its maximum capacity. When a server utilization reaches a given threshold, CAP-ALARM chooses the next server which is not overloaded, if it exists, or the least loaded server.

Some dispatching policies require information about the server loads. Therefore, we implemented load monitors for the following load metrics: number of active connections, CPU utilization and number of I/O requests issued to the disk. Moreover, several load update strategies have been implemented on the dispatcher modules.

The combination of a set of load indexes into a single index that reflects the server load is an interesting research issue that has not yet been investigated in the context of Web clusters. In our test-bed system, we are considering the last five load coefficients coming from each server and we implemented the following three load update strategies:

***Last value*:** this is the most obvious load update strategy. The last load coefficient coming from the server is used as a server state information.

***Arithmetic mean*:** the state load of a server is computed as the arithmetic mean of the last three load coefficients coming from that server.

***Weighted mean*:** the server state information is computed as the weighted mean of the five most recent load samples coming from the server. Each value is combined with a weight obtained by a decay distribution. We use $L_i = \sum_{j=1}^{5}(1/G)l_i(j)e^{-j/2}$, where $l_i(j)$ for $j \in \{1, \ldots, 5\}$ are the load samples of the server i evaluated in the past five intervals of 5 seconds each, and $G = \sum_{j=1}^{5} e^{-j/2}$ is the normalizing factor.

In addition to the choice of the server load index, all server state aware policies face the problem of updating load information. The intervals between load index updates need to be chosen carefully to make sure that the system remains stable. If the interval is too long, performance may be poor because the system is responding to old information about the server loads. On the other hand, too short intervals may result in system overreaction and instability. A strategy for interpreting stale load information that can apply to layer-4 Web switches has been proposed in [54]. Space limitations do not allow us to address this important issue in this paper.

10.2 Workload model

The Web cluster is subject to a mixed workload consisting of both static and dynamic documents. Static pages and objects are served by the Web servers, while dynamic content is processed by the back-ends. Dynamic workload is obtained by means of PHP scripts which are passed to the back-end servers. The same dynamic service may yield very different results depending on when it is requested. This affects response times tremendously. 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 we do not want to perform stress-test analysis, but a fair policy comparison. We have considered four classes of dynamic services.

***Type 1*:** emulates the retrieval of data which is cached at the back-end and does not require intensive CPU or disk usage. The PHP script fills a random size buffer with random numbers and returns it.

***Type 2*:** emulates a query to a database. The corresponding PHP script fills a random size buffer with random numbers, writes it to disk and returns it.

***Type 3*:** emulates an activity that stresses the CPU, such as cyphering. The corresponding PHP script fills a random size buffer with random numbers, cyphers it and returns it.

***Type 4*:** emulates a heavy Web transaction that requires a database query and cyphering of the results. The corresponding PHP script fills a random size buffer with random numbers, ciphers it, writes it to disk and returns it.

In the experiments we compare the performance of dispatching strategies under 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 interarrival times are exponentially distributed with mean equal to 10 seconds and 1 second, respectively. These load mixes do not overload the Web switch and the Web servers, but tend to stress the back-end servers of the test-bed prototype. This is inline with the purposes of this paper that is more oriented to investigate dispatching at a second layer. We have also verified that a different mix of dynamic requests does not change the results significantly, provided that the bottleneck remains at the back-end level and not at the Web switch or at the Web server layer.

In order to emulate realistic Web workloads, we have also considered concepts like user sessions, user think-time, embedded objects per Web page, realistic file sizes and popularity. Table 10.2 summarizes the probability mass function (PMF), the range, and the parameter values of the 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	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(ln x-\mu)^2}{2\sigma^2}}$	$x > 0$	$\mu = 8.215$, $\sigma = 1.46$

Table 10.2: Workload model for static requests and client behavior.

The performance metric of interest for this paper is the response time of a client request, consisting of the base HTML page and all embedded objects. Web workloads are characterized by heavy-tailed distributions, thus response times may assume highly variable values with non-null probability. We have verified, as other authors did in advance, that the mean is not at all representative of the system behavior. Hence, we have preferred to use the 90-percentile of the response time of a client request. This metric is able to represent much better the different behaviors of the dispatching policies examined in the next section.

Because of this choice, we have also modified a synthetic workload generator because none of those available were able to return the 90-percentile of the response time. In particular, for the experiments we use a modified version of the *httperf* benchmark [91] (version 0.8). Client requests are obtained from a pre-generated trace whose characteristics follow the model described above. This approach guarantees also the perfect replicability of the tests that is fundamental for a fair comparison of the performance of the dispatching policies.

10.3 Experimental results

In this section, we present the results of several dispatching algorithms with a focus on the the second-layer. We indicate the solutions that use a centralized policy at the Web switch layer and a distributed algorithm at the Web server layer through the <policy1>/<policy2> notation.

We first consider the class of server state-aware policies to the purpose of showing the difficulty of using server load information for dispatching. Next, we analyze server state-blind policies which do not take into account any kind of server load information, and we verify that these algorithms perform generally better than server state-aware policies. The final goal is to compare centralized and distributed solutions.

10.3.1 Instability of server state information

Various issues need to be addressed when we consider dispatching policies based on some server state information: first of all, the choice for one or more *server load index(es)*; then the way to compute the load state information and the frequency of the samples; finally, due to the cluster architecture, most indexes are not immediately available at the dispatcher, so we have to decide how to transmit them and how frequently.

The focus of this section is on distributed dispatching at the second layer that is, between the Web and the back-end servers. We consider the LARD centralized policy at the Web switch that performs well for static documents. We compare nine server state-aware policies that come as variants of the basic Least Loaded algorithm. In particular, we consider three load indexes evaluated on the back-end servers (that is, LL-CPU, LL-CONN, LL-CONN), and the three metrics “instantaneous load”, “aritmetic mean” (AM) and “weighted mean” (WM), because we are also interested to analyze the impact on performance of using present and/or past samples. For comparison reasons, we finally report the results of another state-aware policy, CAP-ALARM, that uses load information just for skipping overloaded back-end servers.

There are various ways for reading the results reported in the Figures 10.1- 10.6. (Two prelimary observations are in order: the figures on the left side refer to the light workload scenario and response time scales are between 0 and 8 seconds; the figures on the right side refer to the heavy workload scenario and response time scales are between 0 and 12 seconds).

The first important conclusion is a confirmation of the importance of the load index and update interval. Any of these choices has a strong impact on final performance to the extent that the same server LL policy can behave much better or much worse of other algorithms depending on an adequate selection of the load indexes and of the update interval. The dependency on these choices is impressive: the 90-percentiles of response times go from 2.3 to 7 seconds for the light scenario, and from 6 to 12 seconds for the heavy scenario.

If we focus on the sample analysis, we can see that in most instances, even if some instability exists, the shortest update interval (5 seconds) gives the best results. Moreover, the lowest response time is achieved when the last sample or the weighted samples (WM) are considered. On the other hand, the arithmetic mean (AM) that gives the same importance to all samples never performs as the best. These results indicate that it is preferable to make dispatching decisions based on the most recent load information only. The policies which use more past information (e.g., AM) have very poor performance, while algorithms that use the last load sample or give a minor weight to the less recent information (e.g., WM), are preferable. The reason for the ineffectiveness of policies based on more historical load information can be attributed to the dynamics of the multi-tier Web cluster system. Due to the large variability of client requests assigned to a server, the load information becomes obsolete quickly and is poorly correlated with future server load.

From the six figures we can also derive other interesting conclusions. The CPU utilization (LL-CPU) is undoubtely the best index for representing the load on the back-end servers. The curves in Figures 10.3 and 10.4 are the most stable, to the extent that this algorithm is almost no sensitive to the update intervals and to present and past samples. Stability of the results is of primary importance for Web systems subject to highly variable traffic. Moreover, LL-CPU gives the lowest response times with respect to LL-CONN and LL-DISK that is, always below 3 seconds and below 7 seconds for the light and heavy scenario, respectively.

The number of active TCP connections (LL-CONN) is not a good measure of the server load, because almost anything can be requested to a back-end server over a TCP connection, from a small document, possibly cached, to a stressing query. This result was actually expected, while the bad performance of the LL-DISK resulted more surprising. As many queries to the back-end servers are also disk bound, we expected that the number of I/O requests issued in the last update interval would be a good measure of the back-end load. On the other hand, Figures 10.5 and 10.6 show that the 90-percentiles of response times are much higher than those achieved by the

LL-CPU, and comparable, although more stable, to those obtained by the LL-CONN. We give the following motivation to this result. Counting the number of I/O requests suffers from a drawback similar to that discussed about the number of active TCP connections. An access to disk does not reveal much about the computational weight of the involved query. Moreover, as the update interval increases this measure tends to become inaccurate, especially if I/O requests are concentrated in bursts during short periods of time.

We finally compare the results of the LL policies against those of the CAP-ALARM algorithm that does not use server load information to take dispatching decisions but just for alerting the dispatcher to skip an overloaded or unreachable server. Whereas CAP-ALARM performs similarly to LL-CPU when the load is light, for the heavy scenario we have that simply tracking the heavily loaded servers reduces the response times of at least 50% (Figure 10.4). This result is important because this performance improvement can be achieved at a little increase of the communication overheads. Actually, for a fair comparison, this implementation of CAP-ALARM is based on a periodic exchange of state information between each back-end server and Web server similarly to that used for the other server state-aware policies. However, the performance of the CAP-ALARM algorithm could be easily improved by an asynchronous communication mechanism that exchanges alarm messages only when necessary. Similarly to [61], we can conclude that in a distributed system the threshold policy using a small amount of state information provides substantial performance improvement. On the other hand, the LL strategy, which tends to assign every dynamic request to the same back-end for the entire update interval can easily unbalance the server loads and can possibly overload the selected server even for light scenarios.

An important final observation is in order. Although the quantitative values of the performance results are by no means independent of the considered workload model, the relative conclusions about the load indexes, metrics and sample are rather stable.

10.3.2 Performance of server state-blind policies

In this section we consider the policies which do not take into account any server state information for dispatching. There are two classes of server state-blind policies: pure state-blind (e.g., RR) and client state-aware algorithms (e.g., PART, CAP). The figures report also the best server state-aware policy (LLopt) whose performance must be considered as an optimal target which is absolutely not guaranteed for all workload conditions.

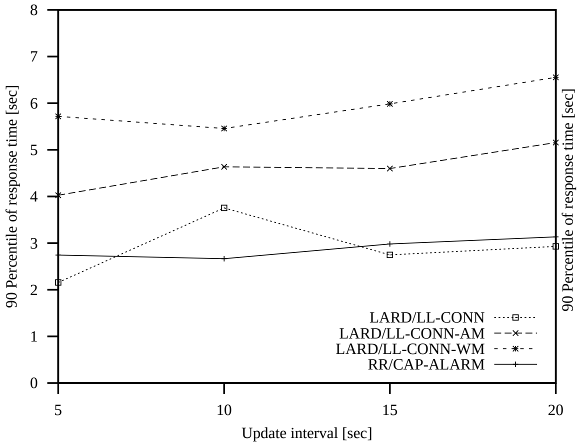


Figure 10.1: LARD/LL-CONN algorithms (*Light scenario*)

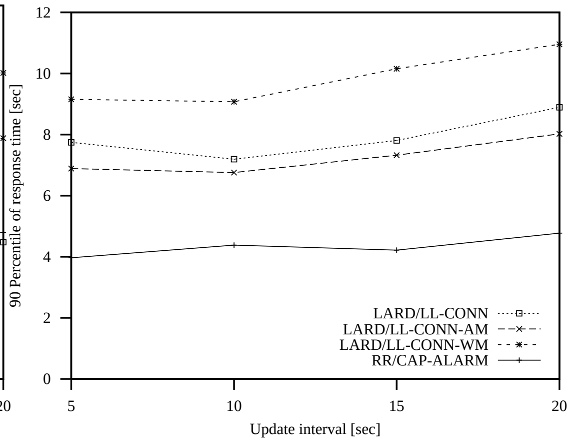


Figure 10.2: LARD/LL-CONN algorithms (*Heavy scenario*)

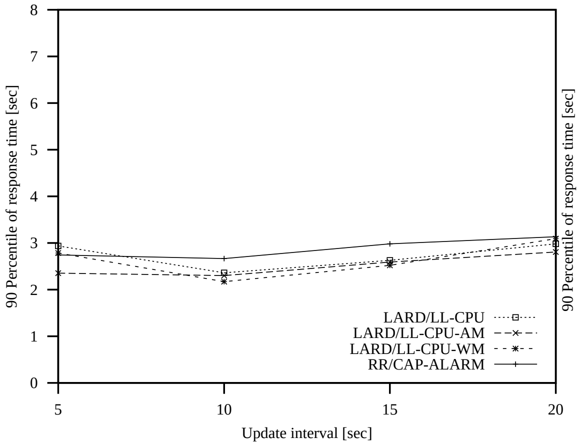


Figure 10.3: LARD/LL-CPU algorithms (*Light scenario*)

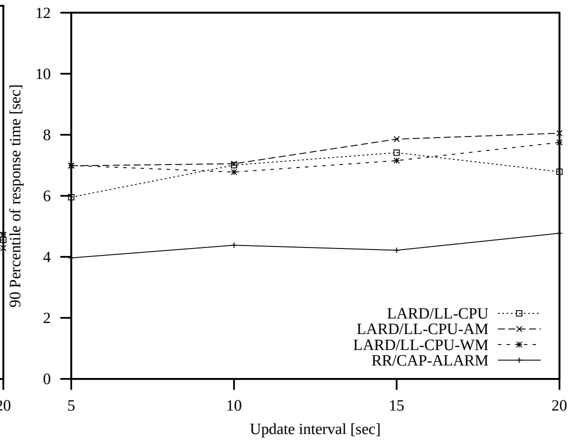


Figure 10.4: LARD/LL-CPU algorithms (*Heavy scenario*)

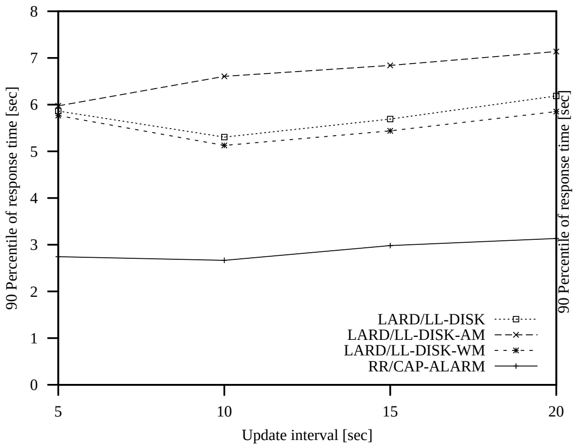


Figure 10.5: LARD/LL-DISK algorithms (*Light scenario*)

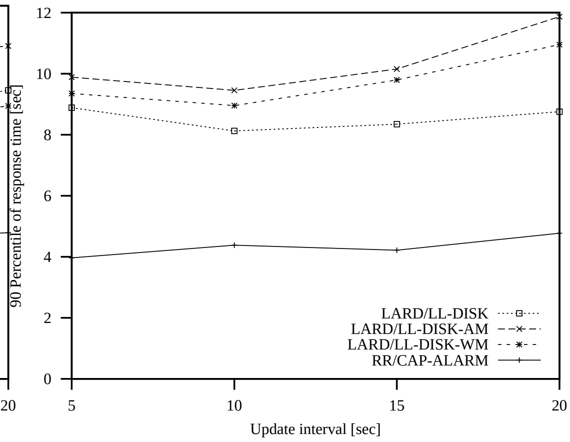


Figure 10.6: LARD/LL-DISK algorithms (*Heavy scenario*)

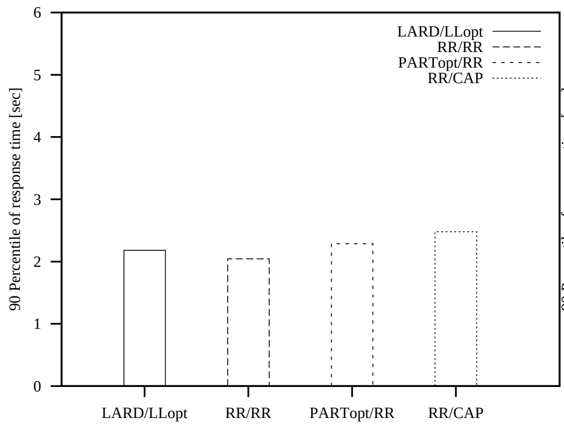


Figure 10.7: State-blind and state-aware policies (*Light scenario*)

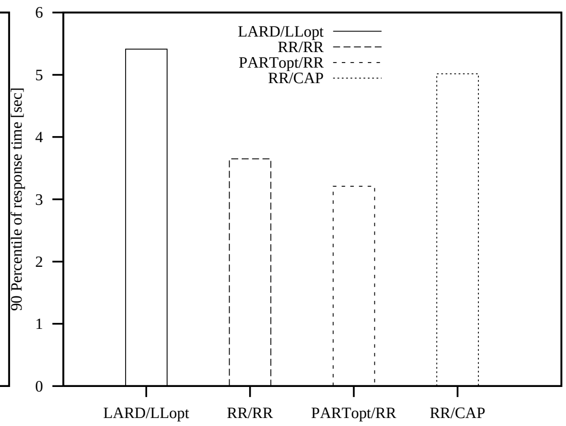


Figure 10.8: State-blind and state-aware policies (*Heavy scenario*)

Figures 10.7 and 10.8 show the 90-percentile of the response time in the case of a light and heavy scenario, respectively. When the load is low, all dispatching policies perform similarly. When we pass to consider a heavier dynamic workload, the performance results of RR/RR and PARTopt/RR are clearly the best. The slight better results of PARTopt/RR over RR/RR were expected because the document partition was done on the basis of a known workload, so to maximize load sharing. This is an optimal condition for the PART policy because different workload distributions could lead the PART policy to perform much poorerly than PARTopt, if requests tend to be polarized to certain Web document directories.

On the other hand, the good results of RR/RR, even better than RR/CAP, were quite surprising (and, we can add, disappointing, because they contradicted the results obtained when RR and CAP were applied to a centralized Web switch [39]). We give the following motivation. It is not convenient to use a multi-class dispatching list for each Web server when there are multiple independent dispatchers, because this augments the probability that two or more requests of the same class are sent to the same back-end server thus overloading it. Instead, dispatching the requests in a plain RR way (that is, through a single class list for each independent dispatcher) limits the risks of conflicts that instead the CAP policy tends to augment.

10.3.3 Centralized vs. distributed algorithms

After the analysis of several distributed policies, we study the performance of a centralized algorithm for dispatching requests to the back-end servers. There are two main ways for implementing a centralized mechanism: to select a master node as in [116] or to let the Web switch decide for both first- and second-layer dispatching. In this paper we refer to the latter choice, because we verified that a query to an external master node for each request dispatching slows down the response time.

Hence, we consider the CAP policy implemented at the Web switch, namely CAP-CENTR, that parses the URL and classifies the type of the request (we recall that there are four classes of dynamic requests and one of static requests). The static requests are assigned to the Web servers in a RR way, while for each dynamic request the Web switch selects a back-end server on the basis of the CAP multi-class list.

Figure 10.9 and 10.10 show the 90-percentile of the response times for the CAP-CENTR algorithm and other distributed algorithms that show the best results. (It should be noted that two of them, that is LARD/LLopt and PARTopt/WRRopt, are optimal versions of the considered policies, for which the shown results are not always guaranteed.) Again, the light scenario (Figure 10.9) does not allow to see appreciable difference.

In a heavier load scenario, the convenience of a centralized (CAP-CENTR) and a state-blind solution (RR/RR) over distributed server state-aware dispatching becomes evident. Indeed, with a distributed mechanism, the dispatchers tend to influence each other negatively, unless some sort of back-end partitioning scheme is adopted. We recall that the PARTopt results shown in these figures are by far an optimal choice that is not guaranteed for other workload conditions.

10.4 Significance of this performance study

Systems with multiple nodes are the leading architectures to build highly accessed Web sites that have to guarantee scalable services and to support ever increasing request load. The performance problems of Web-based architectures is worsening also because of the need of client authentication and system security, the increased complexity of middleware and application software, and the high availability requirements of corporate data centers and e-commerce Web sites. In this paper, we analyze dispatching algorithms that are suitable for multi-tier distributed Web systems. We have analyzed the efficiency and the limitations of different centralized and distributed

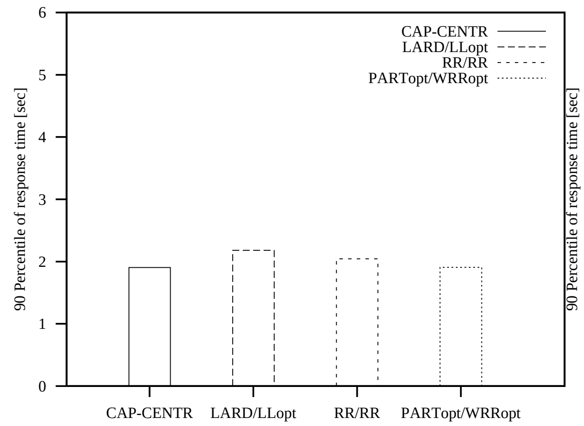


Figure 10.9: Centralized *vs* distributed policies (*Light scenario*)

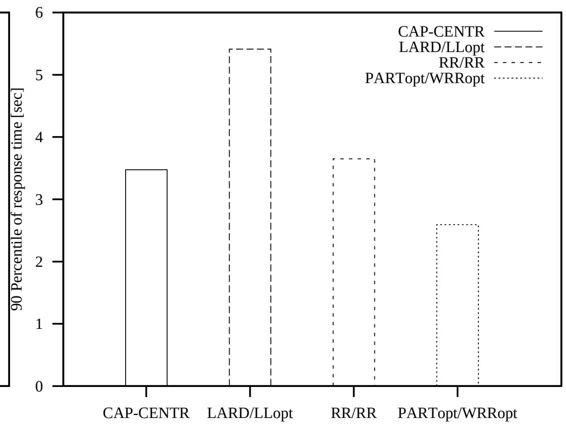


Figure 10.10: Centralized *vs* distributed policies (*Heavy scenario*)

policies and evaluated the tradeoff among the considered alternatives. From the performance study carried out in this paper we can take several conclusions.

- To balance the load across the nodes organized in multiple layers, a Web cluster has two control knobs: the dispatching policy carried out by the Web switch for selecting a Web server for static requests and a dispatching mechanism for selecting a back-end server for dynamic requests. Just exploring one dispatching component alone is inadequate to address uneven distributions of the traffic reaching a Web site.

- Among the most used dispatching algorithms in commercial Web clusters, we found that Round-Robin gives good results and is quite stable, while the Least-Loaded policies are inadequate because they tend to drive servers to saturation as all requests are sent to the same server until new information is propagated. This "herd effect" is well known in distributed systems [54, 90], yet it is surprising that the Least-Loaded approach is commonly proposed in commercial products. Server state-blind algorithms make not so poor assignment decisions as we and other authors expected.

- An initially unexpected result for the server state-aware policies is that the best performance is achieved by algorithms that use only limited state information for checking whether a server is overloaded or not. On the other hand, detailed load information, especially the less recent load information, does not appear useful, and can often lead to bad dispatching choices. The most promising server state-aware algorithms use asynchronous alarms (when the load at a server crosses a threshold) combined with some client information. However, server state load remains an unreliable data which is costly to get and difficult to manage adequately, because it is highly sensitive to the instantaneus state conditions.

- Pure client state-aware policies have a great advantage over policies that use also server information, as they do not require expensive and hard-to-tune mechanisms for monitoring and evaluating the load on each server, gathering the results, and combining them to make dispatching decisions. However, even client state-aware policies should take into account at least a binary server information in order to avoid routing the requests to temporarily unavailable or overloaded servers.

- Static partitioning algorithms tend to ignore load sharing, as it is almost impossible to partition the Web services in such a way that client requests are balanced out. On the other hand, if these algorithms are combined with dispatching policies that guarantee high cache hit rates, such as LARD, they give best results for Web sites providing static information and some simple database information.

Chapter 11

Conclusions

Systems with multiple nodes are the leading architectures to build highly accessed Web sites that have to guarantee scalable services and to support ever increasing request load to Web sites.

In this thesis, we have analyzed routing mechanisms and dispatching algorithms that are suitable for locally distributed Web systems. We have proposed an original taxonomy of the architectures, routing mechanisms and dispatching algorithms. Based on this material, we have proposed new algorithms and analyzed the efficiency and limitations of the different techniques.

The main contributions of this thesis are outlined below.

- Analysis and classification of existing architectures, routing mechanisms and dispatching algorithms for locally distributed Web systems. The original taxonomy allows us to evidence the tradeoffs of the different techniques and the directions for possible improvements.
- A queuing network model of a cluster-based Web system and its solution through a discrete event simulator.
- A Web cluster prototype, called ClubWeb, that permits us to analyze the performance of dispatching mechanisms and algorithms in a real testbed environment. The prototype has been also used to validate the simulation model.
- A new workload model that integrates the most recent results on Web workload characterization for static and dynamic requests, and some secure services that are typically of a modern Web site scenario. This workload model is implemented both as an input for the simulator and as a benchmark tool for the ClubWeb prototype.

- Several content-blind dispatching policies and an entire new class of content-aware dispatching algorithms, named CAP, that are suitable for Web sites subject to highly heterogeneous workload scenarios.

- One of the first studies about dispatching in multi-tier Web cluster architectures.

The Web is becoming the standard interface for accessing remote services and applications, and there is general consensus that Web clusters will be the basic architecture for Web-based information systems, Web hosting centers, and Application Service Providers. Hence, the research interest in this field is likely to increase. This observation is also motivated by the fact that the performance problems of Web-based architecture will tend to worsen because of the proliferation of heterogeneous client devices, the need of client authentication and system security, the increased complexity of middleware and application software, and the high availability requirements of corporate data centers and e-commerce Web sites.

While layer-4 Web cluster architectures may be considered an almost solved problem, the area of content-aware architectures needs further research. Dispatching algorithms that combine effectively client and server information, and adaptive policies are not yet fully explored. Some companies commercialize layer-7 Web switches with very simple dispatching mechanisms that are mainly oriented to a static partition of the Web content and services among the server nodes. Also, the scalability problem posed by layer-7 routing has not been completely solved and non-centralized dispatching algorithms can be a theme of in-depth investigation. A related issue is to avoid state information inconsistency among the multiple dispatchers.

Even more challenging is the study of optimal resource management in multi-tier architectures because the multitude of involved technologies and complexity of process interactions occurring at the middle-tier let the vast majority of commercial products prefer quite naive dispatching algorithms and solutions. Combining load balancing and caching of dynamic content in multi-tier systems is also worth of further investigation.

Another research path is in the direction of combining performance with fault-tolerance and security, and accessibility from different client devices, all topics that are still seen as separate issues in Web clusters. A step further is a Web cluster architecture that gives guaranteed performance and support Quality of Service also on the server side. A significant amount of QoS related research has focused on the network infrastructure, however network QoS alone is not sufficient to support end-to-end QoS. To avoid that high priority traffic may be dropped at the server, the Web-server system should also have mechanisms and policies for delivering end-to-end QoS. Some proposals for providing differentiated service qualities to different classes of users have focused on single-node Web servers and this research topic has been moved towards cluster-based platforms only recently.

The actual improvement of the response time as perceived by the user comes from a combination of technologies, where the multiplication of content provider servers is integrated with geographically dispersed cache servers supported by the content providers themselves or by third-party organizations. Techniques for solving the problems and taking advantage of the potentials originated by the cooperation of multiple servers and multiple caches (e.g., dynamic placement of content, data prefetching, consistency) are still in their infancy, as well as the analysis of the mutual effects of content delivery caching and load distribution.

Bibliography

[1] Adero. `http://www.adero.com`.

[2] Akamai. `http://www.akamai.com`.

[3] Allot Communications Ltd. `http://www.allot.com`.

[4] V. A. F. Almeida, I. M. M. Vasconcelos, and J. N. C. Arabe. The effect of the heterogeneity on the performance of multiprogrammed parallel systems. In *Proc. of Workshop on Heterogeneous Processing*, Beverly Hills, CA, 1992.

[5] E. Anderson, D. Patterson, and E. Brewer. The Magicrouter, an application of fast packet interposing. `http://www.cs.berkeley.edu/~eanders/projects/magicrouter/`, May 1996.

[6] M. Andreolini, E. Casalicchio, M. Colajanni, and M. Mambelli. Performance analysis of layer-7 switches for cluster-based Web servers. Technical Report RR-01.24, Univ. of Rome Tor Vergata, Computer Engineering Dept., Nov. 2001.

[7] M. Andreolini, E. Casalicchio, M. Colajanni, and M. Mambelli. A web-switch for enhancing cluster-based systems with quality of web services. In *Proc. of 11th Int. World Wide Web Conference*, Honolulu, Hawaii, USA, May 2002.

[8] D. Andresen, T. Yang, and O. H. Ibarra. Towards a scalable distributed WWW server on networked workstations. *J. of Parallel and Distributed Computing*, 42:91–100, 1997.

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

[10] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha. Design, implementation and performance of a content-based switch. In *Proc. of IEEE Infocom 2000*, pages 1117–1126, Tel-Aviv, Israel, Mar. 2000.

[11] G. Apostolopoulos, V. Peris, P. Pradhan, and D. Saha. Securing electronic commerce: Reducing the SSL overhead. *IEEE Network*, 14(4):8–16, July/Aug. 2000.

[12] M. F. Arlitt, R. Friedrich, and T. Jin. Workload characterization of a Web proxy in a cable modem environment. *ACM Performance Evaluation Review*, 27(2):25–36, Aug. 1999.

[13] M. F. Arlitt and T. Jin. A workload characterization study of the 1998 World Cup Web site. *IEEE Network*, 14(3):30–37, May/June 2000.

[14] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proc. of ACM Sigmetrics 2000*, pages 90–101, Santa Clara, CA, June 2000.

[15] M. Aron, P. Druschel, and Z. Zwaenepoel. Efficient support for P-HTTP in cluster-based Web servers. In *Proc. of USENIX 1999 Conf.*, pages 185–198, Monterey, CA, June 1999.

[16] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Proc. of USENIX 2000 Conf.*, San Diego, CA, June 2000.

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

[18] L. Aversa and A. Bestavros. Load balancing a cluster of Web servers using Distributed Packet Rewriting. In *Proc. of IEEE Int'l Performance, Computing, and Communication Conf.*, pages 24–29, Phoenix, AZ, Feb. 2000.

[19] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. *World Wide Web*, 2(1-2), 1999.

[20] P. Barford, A. Bestavros, A. Bradley, and M. E. Crovella. Changes in Web client access patterns: Characteristics and caching implications. *World Wide Web*, 2(1-2):15–28, Mar. 1999.

[21] P. Barford and M. E. Crovella. A performance evaluation of Hyper Text Transfer Protocols. In *Proc. of ACM Sigmetrics 1999*, pages 188–197, Atlanta, May 1999.

[22] P. Barford and M. E. Crovella. Critical path analysis of TCP transactions. In *Proc. of ACM Sigcomm 2000*, pages 127–138, Stockholm, Sweden, Aug. 2000.

[23] G. Barish and K. Obraczka. World Wide Web caching: Trends and techniques. *IEEE Communications*, 38(5), May 2000.

[24] T. P. P. C. T.-W. benchmark. `http://www.tpc.org/tpcw`.

[25] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *Integrated Services in the Internet Architecture: an Overview.* RFC 2475, Dec. 1998.

[26] R. Braden, D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: an Overview.* RFC 1633, June 1994.

[27] E. A. Brewer. Lessons from giant-scale services, May 1999. Submitted for publication.

[28] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In *Proc. of USENIX 1998 Conf.*, Berkeley, CA, June 1998.

[29] R. B. Bunt, D. L. Eager, G. M. Oster, and C. L. Williamson. Achieving load balance and effective caching in clustered Web servers. In *Proc. of 4th Int'l Web Caching Workshop*, pages 159–169, San Diego, CA, Apr. 1999.

[30] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven Web sites. In *Proc. of 2001 ACM SIGMOD Int'l Conf. on Management of Data*, 2001.

[31] V. Cardellini, E. Casalicchio, M. Colajanni, and M. Mambelli. Web switch support for differentiated services. *ACM Performance Evaluation Review*, to appear in 2001.

[32] V. Cardellini, E. Casalicchio, M. Colajanni, and S. Tucci. Mechanisms for quality of service in Web clusters. *Computer Networks*, to appear in 2001.

[33] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server system. *acmsurvey*, 2002.

[34] V. Cardellini, M. Colajanni, and P. S. Yu. DNS dispatching algorithms with state estimators for scalable Web-server clusters. *World Wide Web*, 2(2):101–113, July 1999.

[35] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on Web-server systems. *IEEE Internet Computing*, 3(3):28–39, May/June 1999.

[36] V. Cardellini, M. Colajanni, and P. S. Yu. Redirection algorithms for load sharing in distributed Web-server systems. In *Proc. of IEEE 19th Int'l Conf. on Distributed Computing Systems*, pages 528–535, Austin, TX, June 1999.

[37] V. Cardellini, M. Colajanni, and P. S. Yu. Geographic load balancing for scalable distributed Web systems. In *Proc. of IEEE Mascots 2000*, pages 20–27, San Francisco, CA, Aug./Sept. 2000.

[38] E. V. Carrera and R. Bianchini. Efficiency vs. portability in cluster-based network servers. In *Proc. of 8th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, June 2001.

[39] E. Casalicchio and M. Colajanni. A client-aware dispatching algorithm for Web clusters providing multiple services. In *Proc. of 10th Int'l World Wide Web Conf.*, pages 535–544, Hong Kong, May 2001.

[40] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. on Software Engineering*, 14(2):141–154, Feb. 1988.

[41] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic Web data. In *Proceedings of the 19th IEEE Conference on Computer Communications (INFOCOM 2000)*, pages 294–303, New York, NY, Mar. 1999.

[42] J. Challenger, A. Iyengar, P. Dantzig, D. Dias, and N. Mills. Engineering highly accessed Web sites for performance. In Y. Deshpande and S. Murugesan, editors, *Web Engineering*, pages 247–265. Springer-Verlag, 2001.

[43] X. Chen and P. Mohapatra. Providing differentiated service from an Internet server. In *Proc. IEEE Int'l Conf. on Computer Communications and Networks*, Boston, MA, Oct. 1999.

[44] 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.

[45] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving performance of commercial Web sites. In *Proc. of Int'l Workshop on Quality of Service*, London, June 1999.

[46] L. Cherkasova and S. Ponnekanti. Optimizing the "content-aware" load balancing strategy for shared Web hosting service. In *Proc. of IEEE Mascots 2000 Symp.*, pages 492–499, San Francisco, CA, Aug./Sept. 2000.

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

[48] Cisco Systems. DistributedDirector. `http://www.cisco.com/warp/public/cc/pd/cxsr/dd/index.shtml`.

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

[50] M. Colajanni, P. S. Yu, and V. Cardellini. Dynamic load balancing in geographically distributed heterogeneous Web-servers. In *Proc. of IEEE 18th Int'l Conf. on Distributed Computing Systems*, pages 295–302, Amsterdam, The Netherlands, May 1998.

[51] M. Colajanni, P. S. Yu, and D. M. Dias. Analysis of task assignment policies in scalable distributed Web-server systems. *IEEE Trans. on Parallel and Distributed Systems*, 9(6):585–600, June 1998.

[52] Coyote Point. `http://www.coyotepoint.com`.

[53] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Trans. on Networking*, 5(6):835–846, Dec. 1997.

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

[55] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. ONE-IP: Techniques for hosting a service on a cluster of machines. *Computer Networks*, 29(8-13):1019–1027, 1997.

[56] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A middleware system which intelligently caches query results. In *Proc. of ACM/IFIP Middleware 2000*, pages 24–44, Palisades, NY, Apr. 2000.

[57] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of ACM Sigcomm 1989*, Austin, TX, Sept. 1989.

[58] B. Devlin, J. Gray, B. Laing, and G. Spix. Scalability terminology: Farms, clones, partitions, and pack: RACS and RAPS. Technical Report MS_TR-99-85, Microsoft Research, Dec. 1999.

[59] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable and highly available Web server. In *Proc. of 41st IEEE Computer Society Int'l Conf.*, pages 85–92, San Jose, CA, Feb. 1996.

[60] C. Dierks, T. Allen. *The TLS Protocol Version 1.0.* RFC 2246, 1999.

[61] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, 12(5):662–675, May 1986.

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

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

[64] D. Ferrari and S. Zhou. An empirical investigation of load indices for load balancing applications. In *Proc. of Performance 1987*, pages 515–528, North-Holland, The Netherlands, 1987.

[65] Foundry Networks. `http://www.foundrynet.com/`.

[66] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. of 16th ACM Symp. on Operating Systems Principles*, pages 78–91, Saint-Malo, France, Oct. 1997.

[67] S. Gadde, J. Chase, and M. Rabinovich. Web caching and content distribution: A view from the interior. *Computer Communications*, 24(1-2):222–231, Feb. 2001.

[68] X. Gan and B. Ramamurthy. LSMAC: An improved load sharing network service dispatcher. *World Wide Web*, 3(1):53–59, Jan. 2000.

[69] Google Inc. Google inc. http://www.google.com.

[70] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proc. of IEEE 16th Int'l Conf. on Data Engineering*, pages 3–10, San Diego, CA, Apr. 2000.

[71] M. Harchol-Balter, M. E. Crovella, and C. D. Murta. On choosing a task assignment policy for a distributed server system. *J. of Parallel and Distributed Computing*, 59:204–228, Sept. 1999.

[72] C. Huitema. Network vs. server issues in end-to-end performance. Keynote speech at Performance and Architecture of Web Servers 2000, Santa Clara, CA. http://kkant.ccwebhost.com/PAWS2000/huitema_keynote.ppt.

[73] G. S. Hunt, G. D. H. Goldszmidt, R. P. King, and R. Mukherjee. Network Dispatcher: A connection router for scalable Internet services. *Computer Networks*, 30(1-7):347–357, 1998.

[74] IBM. IBM Network Dispatcher. http://www.ibm.com/software/network/dispatcher.

[75] Intel. Intel NetStructure. http://www.intel.com/netstructure.

[76] V. Kanodia and E. W. Knightly. Multi-class latency-bounded Web services. In *Proc. Int'l Workshop on Quality of Service*, Pittsburgh, PA, June 2000.

[77] O. Kremier and J. Kramer. Methodical analysis of adaptive load sharing algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):747–760, Nov. 1992.

[78] D. Krishnamurthy, M. Litoiu, and J. Rolia. Performance stress conditions and capacity planning for e-business applications. In *Proc. of International Symposium on Electronic Commerce*, Beijing, People's Republic of China, May 1999.

[79] D. Krishnamurthy and J. Rolia. Predicting the QoS of an electronic commerce server: Those mean percentiles. In *Proc. of Workshop on Internet Server Performance*, Madison, WI, June 1998.

[80] J. F. Kurose and K. W. Ross. *Computer Networking: A top down approach featuring the Internet*. Addison-Wesley Longman, 2000.

[81] E. Levy-Abegnoli, A. Iyengar, J. Song, and D. Dias. Design and performance of a Web server accelerator. In *Proc. of IEEE Infocom 1999*, pages 135–143, New York, NY, Mar. 1999.

[82] Linux Virtual Server project. http://www.linuxvirtualserver.org/.

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

[84] M.-Y. Luo and C.-S. Yang. Constructing zero-loss Web services. In *Proc. of IEEE Infocom 2001*, Anchorage, Alaska, Apr. 2001.

[85] M.-Y. Luo and C.-S. Yang. System support for scalable and reliable and highly manageable Web hosting service. In *Proc. of USENIX Symposium on Internet Technology and Systems*, San Francisco, CA, Mar. 2001.

[86] A. M. Luotonen. *Web proxy servers.* Prentice Hall, 1997.

[87] D. A. Menascé, J. Almeida, R. Fonseca, and M. A. Mendes. Business-oriented resource management policies for e-commerce servers. *Performance Evaluation*, 42(2-3):223–239, Sept. 2000.

[88] Mesquite Software Inc. *CSIM18 user guide.* Austin, TX. `http://www.mesquite.com/`.

[89] Mindcraft. WebStone benchmark. `http://www.mindcraft.com/`.

[90] M. Mitzenmacher. How useful is old information. *IEEE Trans. on Parallel and Distributed Systems*, 11(1):6–20, Jan. 2000.

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

[92] NetScaler Inc. Netscaler's Request Switch. `http://www.netscaler.com`.

[93] Nortel Networks. Nortel Networks Web OS. `http://www.nortelnetworks.com/`.

[94] Oracle Inc. Oracle i-cache. `http://www.oracle.com/`.

[95] O. Othman, C. O'Ryan, and D. C. Schmidt. Strategies for CORBA middleware-based load balancing. *IEEE Distributed Systems Online*, 2(3), Mar. 2001.

[96] 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 *Proc. of 8th ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, CA, Oct. 1998.

[97] 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.

[98] R. Pandey and R. Barnes, J. F. Olsson. Supporting quality of service in HTTP servers. In *Proc. of ACM Symp. on Principles of Distributed Computing*, pages 247–256, Puerto Vallarta, Mexico, June 1998.

[99] C. Perkins. *IP encapsulation within IP.* RFC 2003, Oct. 1996.

[100] J. E. Pitkow. Summary of WWW characterizations. *World Wide Web*, 2(1-2):3–13, Mar. 1999.

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

[102] Resonate Inc. `http://www.resonate.com/`.

[103] D. Rosu, A. Iyengar, and D. Dias. Web proxy acceleration. *Cluster Computing*, to appear in 2001.

[104] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[105] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, Dec. 1992.

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

[107] J. Song, E. Levy-Abegnoli, A. Iyengar, and D. Dias. Design alternatives for scalable Web server accelerators. In *Proc. of 2000 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pages 184–192, Austin, TX, Apr. 2000.

[108] P. Srisuresh and D. Gan. *Load sharing using IP Network Address Translation*. RFC 2931, 1998.

[109] 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.

[110] J. Wang. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review*, 29(5):36–46, Oct. 1999.

[111] Y. T. Wang and R. J. T. Morris. Load sharing in distributed systems. *IEEE Trans. on Computers*, 34(3):204–217, Mar. 1985.

[112] D. Wessels. *Web caching*. O'Reilly and Associates, Cambridge, MA, 2001.

[113] C.-S. Yang and M.-Y. Luo. A content placement and management system for distributed Web-server systems. In *Proc. of 20th IEEE Int'l Conf. on Distributed Computing Systems*, pages 691–698, Taipei, Taiwan, Apr. 2000.

[114] Zeus Technologies Ltd. Zeus Web server. `http://www.zeustechnologies.com`.

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

[116] H. Zhu, B. Smith, and T. Yang. Scheduling optimization for resource-intensive Web requests on server clusters. In *Proc. of 11th ACM Symp. on Parallel Algorithms and Architectures (SPAA'99)*, pages 13–22, June 1999.

[117] H. Zhu, H. Tang, and T. Yang. Demand-driven service differentiation in cluster-based network servers. In *Proc. of IEEE Infocom 2001*, Anchorage, Alaska, Apr. 2001.