Università degli Studi di Roma “Tor Vergata”

Scalable Web-server Systems

Valeria Cardellini

Doctoral Dissertation in
“Computer Science and Control Engineering”

Advisors:
Prof. Michele Colajanni
Prof. Salvatore Tucci

PhD Coordinator:
Prof. Daniel Pierre Bovet

Committee in charge:
Prof. Giuseppe Serazzi, Chair
Prof. Bruno Ciciani
Prof. Floriana Esposito

Roma, Marzo 2001
Scalable Web-server Systems

Copyright 2001

by

Valeria Cardellini
Acknowledgments

This thesis would not have been possible without the continuous support, guidance, and friendship of many people, who I would like to thank for this opportunity.

First, I would like to thank my advisors, Prof. Salvatore Tucci and Prof. Michele Colajanni, for their invaluable guidance, their patience, their examples, and their tireless efforts in shaping my technical abilities. I am grateful to Prof. Tucci who introduced the distributed systems field to me, and initiated and supported my interest and passion in doing research in this area. Michele is a great advisor and I greatly benefited from his constant support, guidance, and inspiration during these years.

My sincere thanks to Dr. Philip S. Yu for his excellent technical guidance and fruitful collaboration. I would also like to thank him for finding time to guide my research while I was at IBM T.J. Watson Research center. His technical advice has been extremely valuable to me.

I am also grateful to my graduate colleagues, who have made my stay at Tor Vergata an enjoyable experience. In particular, I would like to thank Emiliano Casalicchio and Fabio Massimo Zanzotto.

Lastly, but most importantly, I would like to thank my parents, my sister Marina, and Cesare for their continuous encouragement throughout these years. My parents have always given me love, selfless care, and support. Their passion for career and dedication to work have also been role models to me. This thesis is dedicated to them.
# Contents

1 **Introduction** 1
   1.1 The Need for Scalable Web-server Systems 1
   1.2 Issues in Scalable Web-server Systems 3
   1.3 Thesis Contributions 5
   1.4 Thesis Outline 7

2 **Background** 9
   2.1 Analysis of a Web Transaction 10
   2.2 A Brief Overview of DNS 12
      2.2.1 The Impact of TTL Values 15
   2.3 HTTP Protocol 16
   2.4 Web Servers 19
      2.4.1 Content Provided by Web Servers 21
      2.4.2 Multi-tier Architectures 22
   2.5 Performance Metrics for Web Services 23
      2.5.1 Server-side Performance Metrics 24
      2.5.2 Client-side Performance Metrics 25
   2.6 Multiple Web Servers 27
      2.6.1 Attributes for Web-server Systems 29
      2.6.2 Routing Along the Web Request Path 30
      2.6.3 Solutions not Examined 31

3 **Locally Distributed Web-server Systems** 33
   3.1 System Architectures 34
      3.1.1 Web-server Clusters 35
   3.2 Routing Mechanisms for Web-server Clusters 37
      3.2.1 Layer-4 Solutions 38
3.2.2 Layer-7 Solutions .................................. 42
3.2.3 Routing Mechanisms Comparison ..................... 46
3.3 Dispatching Algorithms for Web-server Clusters ........... 49
  3.3.1 A Taxonomy of Dispatching Algorithms ............... 50
  3.3.2 Content-blind Dispatching .......................... 52
  3.3.3 Content-aware Dispatching .......................... 56
  3.3.4 Dispatching Algorithms Comparison .................. 58
3.4 Classification and Summary of Products and Prototypes .. 61
  3.4.1 Layer-4 Solutions ................................ 61
  3.4.2 Layer-7 Solutions ................................ 63
3.5 Architectures Combining Basic Solutions ................ 65
  3.5.1 Integrated Solutions ............................... 65
  3.5.2 Multi-tier Architectures ............................. 67
3.6 Concluding Remarks ................................ 69

4 Geographically Distributed Web-server Systems ............. 71
  4.1 System Architectures .................................. 73
    4.1.1 Distributed Web-server System .................... 73
    4.1.2 Distributed Web Clusters ......................... 74
  4.2 Routing Mechanisms .................................. 75
    4.2.1 DNS Mechanisms .................................. 76
    4.2.2 Network Mechanisms ............................... 78
    4.2.3 Web System Mechanisms ......................... 79
    4.2.4 Routing Mechanisms Comparison .................. 82
  4.3 Dispatching Algorithms ................................. 85
    4.3.1 A Taxonomy of Dispatching Algorithms .............. 85
    4.3.2 Network Proximity ................................ 86
    4.3.3 DNS-based Dispatching ............................ 88
    4.3.4 Web System Dispatching ............................ 90
    4.3.5 Dispatching Algorithms Comparison ................. 92
  4.4 Classification and Summary of Products and Prototypes .... 93
    4.4.1 Distributed Web-server Systems .................... 93
    4.4.2 Distributed Web Clusters ........................... 95
  4.5 Content Distribution Networks .......................... 97
  4.6 Concluding Remarks ................................ 99
5 Load Sharing through DNS-based Mechanisms .................................................. 101
  5.1 DNS-based Web System ................................................................................. 103
     5.1.1 System Environment ............................................................................. 103
     5.1.2 State Information for DNS Dispatching .................................................. 104
  5.2 DNS Algorithms with Constant TTL .......................................................... 105
  5.3 DNS Algorithms with Adaptive TTL .......................................................... 107
     5.3.1 Probabilistic Algorithms ....................................................................... 108
     5.3.2 Deterministic Algorithms ...................................................................... 109
  5.4 Information Gathering Mechanisms ............................................................ 109
     5.4.1 Server State Information ....................................................................... 110
     5.4.2 Domain State Information .................................................................... 111
  5.5 Simulation Model .......................................................................................... 114
  5.6 Experimental Results .................................................................................... 118
     5.6.1 Performance Metrics ............................................................................. 118
     5.6.2 Performance of Constant TTL Algorithms ............................................. 119
     5.6.3 Comparison of Constant and Dynamic TTL Algorithms ...................... 120
     5.6.4 Sensitivity to System Heterogeneity ...................................................... 123
     5.6.5 Sensitivity to TTL Values and Workload Parameters ............................ 125
     5.6.6 Robustness of Adaptive TTL Algorithms ............................................. 127
     5.6.7 Summary of the Performance Study ...................................................... 130
  5.7 Concluding Remarks ...................................................................................... 132

6 Load Sharing through Server-based Mechanisms .............................................. 133
  6.1 Design Space for Redirection Schemes ......................................................... 135
  6.2 Synchronous Redirection Algorithms ......................................................... 137
     6.2.1 Domain Redirection ............................................................................... 139
     6.2.2 Client Redirection .................................................................................. 140
     6.2.3 Domain and Client Redirection ............................................................. 142
     6.2.4 Alarm Messages .................................................................................... 142
  6.3 Asynchronous Redirection Algorithms ......................................................... 143
  6.4 Simulation Model .......................................................................................... 144
  6.5 Experimental Results ..................................................................................... 147
     6.5.1 Performance Metrics ............................................................................. 148
     6.5.2 Performance of Synchronous Redirection Algorithms .......................... 149
     6.5.3 Performance of Asynchronous Redirection Algorithms ....................... 153
6.5.4 Sensitivity to System Parameters .......................... 155
6.5.5 Sensitivity to Workload Model ............................ 156
6.6 A Qualitative Comparison to Related Work .................. 158
6.7 Concluding Remarks ........................................ 159

7 Minimizing User Response Time ............................... 161
  7.1 DNS-based Dispatching Algorithms .......................... 163
  7.2 Server Redirection Algorithms .............................. 165
  7.3 Simulation Model .......................................... 167
    7.3.1 System and Workload Model ........................... 167
    7.3.2 Network Model ....................................... 171
  7.4 Experimental Results ..................................... 174
    7.4.1 Performance Metrics ................................. 175
    7.4.2 Static Scenario ..................................... 175
    7.4.3 Dynamic Scenario .................................... 181
  7.5 Concluding Remarks ........................................ 187

8 Conclusions .................................................... 189
  8.1 Summary of Contributions .................................. 189
  8.2 Future Directions ........................................ 191

Bibliography ..................................................... 193
Chapter 1

Introduction

1.1 The Need for Scalable Web-server Systems

In the past few years, the overwhelming success of the World Wide Web has placed dramatic pressure on the Internet infrastructure. The increasing service demand, coupled with the growing importance of electronic commerce, has exasperated the need for high performance, scalable, and available Web systems capable of serving today’s 200 million Web users. The “first generation Web” was a channel for non critical information, where 90% of information consisted of static content, such as text and images. Instead, the “second generation Web” provides services by far complex than those related to Web publishing. The second generation Web is characterized by a growing percentage of dynamically generated content and has become an important channel for critical and secure information. Moreover, Web-related software is getting a fundamental technology for information systems of the most advanced companies and organizations.

Many users already rely on the Web for up-to-date personal and business information and transactions. Users do not know nor care of complexity of Web infrastructure and technology. They do complain if the response time becomes too long (e.g., they are not willing to accept response times greater than eight-ten seconds), there are too many periods of unavailability, the security of transactions is not fully guaranteed. That is to say, they would like to see a satisfactory service performance at all times. Substantial changes are transforming the Web from a communication and browsing infrastructure into a medium for conducting personal businesses and e-commerce, so demands placed on Web services continue to grow and Web sites are becoming more stressed than ever.
Many factors impact on Web site performance. For instance, performance problems may occur at network level either because of congested Internet backbones or due to long network routes; they may happen as well as at server level either because of underprovisioned capacity or due to an unexpected surge of requests. Although in recent years both network and server capacity have improved, application protocols have been enhanced, and new architectural solutions have been developed, the issue of Web performance continues to challenge researchers. As a consequence, the need to optimize Web-based services is producing a variety of novel architectures for high performance networks and servers.

In this thesis, we focus on the server side being our main interest to reduce the Web server delay, which is the response time component under the control of Web site’s designers. Indeed, the Web performance perceived by users is increasingly dominated by server delays, especially when contacting busy servers [25]. Recent measures suggest that the Web servers account for about 40% of the delay in a Web transaction [78] and it is likely that this percentage will go up in the near future, as improvement in network bandwidth will pose more strain on Web servers. In this scenario, the brute-force approach of adding ever increasing network and server capacities will not solve the Web scalability problem in a foreseeable future.

This thesis is concerned with Web-server systems composed by multiple nodes, which are the only viable and effective solution to provide Web services that keep up with the increasing performance demands. Indeed, single node platforms that do not replicate information content cannot provide the scalability needed to handle large traffic volumes and satisfy users’ expectations. A common approach, adopted by highly accessed Web sites to handle millions of requests per day, is to rely on a Web-server system that transparently dispatches requests among its nodes, thus appearing as a single virtual service to users.

To highlight the need for scalable Web-server systems, it may help quantify the magnitude of scale of Web traffic handled by sites that receive a large number of requests. To this purpose, we report in Table 1.1 some data about the volume of Web traffic managed by some of the most highly accessed sites during the last few years. The sharp and endless increase in the traffic volume leaps to the eye. Indeed, we observe that, while in 1995 the peak load was about 2 million of requests on a single day, at the end of 2000 about 1,2 million of requests were addressed to a Web site during a single peak minute. We also notice that the world record for volume has been surpassed time and time again.
Table 1.1: Highly accessed Web sites (load measures are in requests).

<table>
<thead>
<tr>
<th>Event</th>
<th>Period</th>
<th>Peak day</th>
<th>Peak minute</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCSA server (Oct. 1995)</td>
<td>-</td>
<td>2 Million</td>
<td>-</td>
</tr>
<tr>
<td>Olympic Summer Games (Aug. 1996)</td>
<td>192 Million (17 days)</td>
<td>8 Million</td>
<td>-</td>
</tr>
<tr>
<td>Presidential US Election (Nov. 1996)</td>
<td>-</td>
<td>9 Million</td>
<td>-</td>
</tr>
<tr>
<td>NASA Pathfinder (July 1997)</td>
<td>942 Million (14 days)</td>
<td>40 Million</td>
<td>-</td>
</tr>
<tr>
<td>Olympic Winter Games (Feb. 1998)</td>
<td>634.7 Million (16 days)</td>
<td>57 Million</td>
<td>110,000</td>
</tr>
<tr>
<td>Wimbledon (July 1998)</td>
<td>-</td>
<td>-</td>
<td>145,000</td>
</tr>
<tr>
<td>FIFA World Cup (July 1998)</td>
<td>1,352 Million (84 days)</td>
<td>73 Million</td>
<td>209,000</td>
</tr>
<tr>
<td>Wimbledon (July 1999)</td>
<td>-</td>
<td>125 Million</td>
<td>430,000</td>
</tr>
<tr>
<td>Wimbledon (July 2000)</td>
<td>-</td>
<td>282 Million</td>
<td>964,000</td>
</tr>
<tr>
<td>Olympic Summer Games (Sept. 2000)</td>
<td>-</td>
<td>875 Million</td>
<td>1,200,000</td>
</tr>
</tbody>
</table>

In addition to the data shown in Table 1.1 which refer to occasional highly popular events (e.g., Olympic Games), it is noticeable that some highly accessed Web sites such as Altavista, CNN, Lycos, and Yahoo have to face more than 50 million of requests every day. From such a large number of requests arises the need to host a Web site on multiple Web server nodes that should provide scalable, high performing, and available Web services.

1.2 Issues in Scalable Web-server Systems

In this thesis, we consider a scalable Web-server system to be any architecture consisting of multiple Web-server nodes (administered by the Web site technical management), with a mechanism to spread incoming client requests among the nodes. Scalable Web systems can be broadly classified according to the type of distribution of the server nodes in local distribution, in which the system is composed by a tightly coupled architecture placed at a single location, and global distribution, in which the Web servers are distributed over the Internet. Both systems include a routing mechanism to direct the client request to the target server node, a dispatching algorithm to select the Web server node best suited to respond, and an executor to carry out the dispatching algorithms and support the routing mechanism.

The deployment of scalable Web-server systems poses a unique set of challenges both at architectural and management level. The first key issue concerns the apparently infinite users population that those systems are required to serve, considering also the exponential growth in the client demand curve which characterizes the Web. Supporting a large number of accesses impacts considerably on the system architecture and infrastructure, as any site could be required to respond to millions of requests.
per day with a service of guaranteed quality level, or to scale for matching rapid and
dramatic changes in the number of clients. Indeed, Web site engineers have to face
the sudden arrival of “flash crowds” of users, often causing server overload and net-
work congestion. These sudden spikes in traffic increase the difficulty to size server
capacity adequately and make scalability an essential feature of Web systems.

The second issue arises from the need to keep the distributed nature of the server
systems as transparent as possible to users, so that they do not need to be concerned
about the names or locations of the multiple servers. Therefore, it is necessary to
rely on routing mechanisms that allow to distribute incoming requests transparently
among several server nodes.

A third issue regards the engineering of Web sites through distributed and parallel
technologies. Indeed, cooperative methods, parallel architectures and distributed
systems could easily exacerbate the situation if the system is not well designed for
specific Web services and the results are not evaluated carefully.

After years of proposals of new algorithms and architectural solutions (the first
dates back to 1994 when NCSA server had to face the first million of requests per
day [98]), some aspects of multiple server architectures for Web sites are achieving a
stable position. However, other issues remain to be addressed: the technology, that
was adequate to manage the first generation of Web-server systems, does not scale to
meet the demands created when number of users, as well as complexity and variety of
services, increase dramatically. Mechanisms and algorithms, that were once sufficient,
often fail to scale to the level required or perform slowly and inefficiently.

In this thesis, we analyze and design Web-server systems in which performance
and scalability constitute a primary objective. As load sharing is instrumental in ob-
taining high performance server systems, we propose and evaluate several dispatching
algorithms for Web-server systems. We focus on geographically distributed archi-
tectures as they are the most scalable architectures to handle thousand of requests
per minute that stress highly accessed Web sites. Specifically, in this thesis we con-
sider Web-server systems in which the first-level dispatching occurs at the Domain
Name System (DNS) level, during the translation of the Web site name into a cor-
responding IP address. The popularity of DNS-based architectures to manage global
content distribution is increasing due to their ability to easily scale up, their seamless
integration with the current Internet infrastructure, and the generality of the name
resolution process. Indeed, a large number of commercial products for geographically
distributed Web systems, as well as content distribution networks, rely on DNS-based
mechanisms.

Web systems based on the Domain Name System have, however, to address some challenging issues beyond those common to other approaches for Web-server systems. A main problem arises from the coarse-grained type of control that the dispatching entity has on request assignments; naive solutions to augment the control are not feasible to avoid stressing the DNS infrastructure and degrading response time experienced by users. A second problem stems from the uneven distribution of the users accessing the Web site among the Internet domains; such non-uniformity is amplified in a geographical context by world time zones, by the time of the day as well as by the day of the week. All these issues create interesting challenges to DNS-based Web systems, that will be addressed in the thesis through the deployment and integration of system architectures and dispatching algorithms to be executed by different components of the Web-server system.

We consider first a one-level Web system architecture, where the request assignment occurs only at DNS level and we exploit dispatching algorithms executed by the authoritative DNS server of the Web site which address DNS problems. Then, we focus on two-level and three-level Web system architectures, in which the routing and the dispatching decision can occur at two or three levels in the system. The first-level dispatching is still centralized and executed by the authoritative DNS server, while the second or third dispatching is typically distributed and directly deployed by the Web server nodes through some redirection mechanism. Multi-level dispatching allows to achieve a more fine-grained type of control on request assignments than single-level dispatching, especially in DNS-based Web systems; however, it opens up a new dimension to exploit redirection mechanisms and algorithms and their integration with the first-level assignment decision.

1.3 Thesis Contributions

The goal of this thesis is to analyze and design scalable and high performing Web-server systems. Our basic premise is the compatibility of all proposed solutions with existing Web standards and protocols so that any considered architecture, dispatching algorithm and routing mechanism could be immediately adopted.

Our contributions with this thesis are the following.

- We analyze and classify the existing architectures, routing mechanisms and dispatching algorithms for scalable Web-server systems. The analysis not only
examines and characterizes Web-server systems proposed so far, but also points out the tradeoffs posed by the different techniques and the directions to develop more effective approaches. Moving from the analysis of the existing solutions, we focus on Web-server systems, where the first-level dispatching is based on the Domain Name System.

- We present and evaluate DNS-based dispatching algorithms that address the issue of system load sharing. The proposed algorithms rely on client and server state information accessible to the system dispatching entity and act adaptively on all parameters controlled by the dispatching entity. We demonstrate that the proposed strategies reduce load imbalance effectively, even when the Web-server system is highly heterogeneous and the control on assignment is limited to a very small portion of requests reaching the Web site.

- We propose and evaluate a two-level Web system architecture that ensures system load sharing. We integrate the DNS-based dispatching with some server redirection algorithms, to avoid load fluctuations originating from the first-level assignment. We analyze both coarse- and fine-grained request reassignments and different types of system state information. We show that our solution is effective to overcome sudden surges of load on single server nodes, thus avoiding overload periods even under a realistic and highly skewed workload.

- We propose and evaluate a three-level system architecture and the integration of routing mechanisms and dispatching algorithms, aimed at minimizing user's response time and at achieving load sharing as well. We show that most of the approaches for global distribution taking client proximity into account are not able to satisfy users' expectations. We also demonstrate that well-designed server redirection algorithms enhance considerably the quality of Web services provided to end users in a geographical environment and have the ability to face sudden surges of load, which a geographical environment exasperates.

- We develop a quantitative evaluation of all proposed solutions through system simulation. We design detailed system models and realistic heavy-tailed workload models. We also develop a simple model of the Internet infrastructure, which provides a fair testbed to compare the performance results of the proposed algorithms and mechanisms taking network information into consideration.
1.4 Thesis Outline

In Chapter 2, we present background information instrumental to the rest of the thesis and introduce some terminology that will be used throughout the next chapters. We discuss the major causes of delay that may affect the overall Web object retrieval time, and briefly describe the components that are involved in the execution of a Web transaction and contribute to the delay. Giving motivations for the goal of this thesis, we describe the requirements of scalable server architectures that are considered throughout the thesis and analyzes the points along the Web transaction where the request distribution can occur.

In Chapters 3 and 4, we analyze and classify architectures, routing mechanisms and load sharing algorithms that have been developed so far to design scalable Web-server systems; we also identify some of the issues associated with setting up and managing systems for highly accessed Web sites. Specifically, in Chapter 3 we describe and discuss the various approaches for routing and dispatching requests in locally distributed Web systems, while in Chapter 4 we deal with geographically distributed Web systems.

In Chapter 5, we start investigating geographically distributed Web-server systems focusing on dispatching policies, where the decision on client request assignment occurs at the Domain Name System level during the address lookup phase. We describe and evaluate a new class of load sharing policies that are capable of avoiding Web server overloads and can face the main problems of DNS-based dispatching, particularly the coarse-grained control on client requests reaching the Web-server system, and the uneven client distribution among Internet domains.

In Chapter 6, we present a Web-server system characterized by a double level of requests routing and dispatching. We integrate the coarse-grained DNS dispatching with a request redirection mechanism carried out by the Web servers; then we propose and evaluate a large set of alternative redirection algorithms that aim at achieving load sharing in the two-level dispatching architecture. Our results demonstrate that Web-server redirection mechanisms can effectively distribute the load and handle high request rates, thus minimizing system overload.

In Chapter 7, we investigate a distributed Web-server architecture in which the requests routing and dispatching may occur at three different system levels, that is at the authoritative DNS of the Web site, at the dispatching entity in front of each local Web system, and at each Web server. We also address the key issue of minimizing
the impact of WAN network delays on the response time perceived by users through the investigation of server redirection algorithms that limit the number of necessary redirections.

Finally, in Chapter 8 we summarize results and contributions of this work and indicate some directions for future research on the basis of the results presented in this thesis.
Chapter 2

Background

Web traffic carried over the Hypertext Transfer Protocol (HTTP) accounts for 70-75% of the Internet backbone, according to some recent studies [45, 104, 148]. With the Web being the main application used on the Internet, it is important to understand the possible sources of latency that impact on the response time perceived by end users, so to ensure an efficient use of both network infrastructure and server resources by Web transfers.

This chapter presents background information instrumental to the rest of the thesis and introduces some terminology that will be used throughout the next chapters. We discuss the major causes of delay that may affect the overall Web object retrieval time, and briefly describe the components (that is, Domain Name System servers and Web servers) and protocols (that is, HyperText Transfer Protocol) that are involved in the execution of a Web transaction and contribute to the user-perceived delay. In Section 2.1 the basic components of a Web object access are discussed. Section 2.2 covers background regarding the Domain Name System (DNS) mechanism, as the DNS lookup time may contribute considerably to the retrieval of the Web object. Moreover, DNS-based server selection algorithms to share the load among multiple Web servers will be proposed and evaluated later in this thesis. Section 2.3 summarizes the main characteristics of the HyperText Transfer Protocol and surveys the new features introduced in the most recent protocol version that are of interest to this thesis. Section 2.4 presents the basic operation of Web servers as well as their limitations. Section 2.5 describes the main performance metrics for Web services. Finally, Section 2.6 gives motivations for this thesis, describes the main attributes of scalable server architectures and analyzes the points along the Web transaction where the request distribution can occur.
2.1 Analysis of a Web Transaction

In this section, we analyze the phases for the retrieval of a typical Web object and identify the main delay components that affect user-perceived Web latency.

Before starting the analysis, it is useful to review some terminology related to Web content, that is typically structured using the HyperText Markup Language (HTML) [156]. A Web page is a multi-part document consisting of a collection of objects. A Web object is simply 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.foo.com/index.html). A URL has two main components: the hostname of the server that houses the object (e.g., www.foo.com) and the object’s path name (e.g., index.html). 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. A session is a sequence of Web page requests issued by the same user during a single visit to a Web site.

A Web transaction starts when the end user makes a request to a Web server, by either typing an URL or clicking on a link. 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. In order to access a Web page, a browser issues multiple requests to the Web server to retrieve all the Web objects composing it. So, we refer to a Web transaction as the complete interaction that starts when the client issues a request to a Web server and ends when the client receives the last object of the requested Web page.

Accessing a Web object, e.g., http://www.foo.com/index.html, involves the following basic steps.

1. First, the client extracts the hostname of the Web site (www.foo.com) from the requested URL and sends it to its local name server, in order to resolve the symbolic name to a corresponding IP address. The local name server obtains the IP address by contacting a well-known root of the domain name server hierarchy and ultimately querying foo’s authoritative DNS server. The local name server caches the obtained IP address and returns it to the client.

2. After having received the IP address of the Web server, the client establishes a
3. The client sends the HTTP request for index.html to the Web server. This request follows a router path on the Internet until it reaches the Web server, which sends the requested object back. This step can be repeated for each object embedded within a composite Web page.

Therefore, five basic components of the Web delay can be identified in the retrieval of a single Web object, as shown in Figure 2.1.

a. The **address lookup delay** (also, address resolution delay) is the time needed to map the hostname into a corresponding IP address;

b. The **connection delay** is the time needed to establish the TCP connection (and, eventually, to terminate it at the end of the transaction);

c. The **server delay** is the time needed by the server to schedule the object request, prepare the response, and send information back to the client. This delay includes service time and waiting time at the various resources of the server, such as processor, disk, and network interface card.

d. The **network delay** (also, transmission delay) is the delay needed to deliver data packets on the Internet.

e. The **client delay** is the time spent by the client browser to parse the received response and render it to the user.

The basic sequence shown in Figure 2.1 can be altered by several factors, such as:

- caching of the hostname-to-address mapping at the client browser, which avoids that all HTTP requests are preceded by an address resolution;

- caching of the hostname-to-address mapping at some intermediate cache between the local name server and the authoritative name server;

- version of HTTP protocol being used in the client/server interaction (i.e., support 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.
Some of these factors will be analyzed in the next sections, in which we will review the Domain Name System, HTTP protocol, and Web servers.

2.2 A Brief Overview of DNS

The Domain Name System (DNS) provides a service whose primary function is to translate human readable domain names (e.g., www.foo.com) into numerical IP addresses of corresponding machines (e.g., 160.80.85.38) [3]. It is implemented in a distributed database of records spread across a semi-static hierarchy of servers, called name servers. The name space is partitioned into a hierarchy of domains and subdomains, where each domain represents all names with a given suffix and is administered independently by one (or more) authoritative name server(s); this performs translation for all names with the corresponding suffix. More than one name server can be designated as authoritative for fault tolerance reasons. Typically, name servers responsible for large domains (e.g., .com) delegate other name servers to be in charge of subdomains (e.g., foo.com). Therefore, the resolution of a DNS query may require contacting several name servers that compose the delegation chain for a particular domain name.

Name servers maintain the mapping between hostnames and IP addresses in resource records (RR). Different RRs types exist, as described in [111, 112]. These RFCs
Section 2.2. A Brief Overview of DNS

represent the current specification of DNS. The RR types that are most relevant to this overview are:

- A (Address) record, that contains the standard hostname to IP address mapping (also, referred to as the name-to-address mapping).

- NS (Name Server) record, that defines the hostname of the authoritative name server for a given domain.

- SOA (Start Of Authority) record, that identifies the start of the zone of authority for a given name server. A zone is a point of delegation in the DNS hierarchy; it contains all the domain names from a certain point downward in the hierarchy, except those which are delegated to other zones.

The DNS system scales by caching resource records at intermediate name servers. Each resource record has an associated Time To Live (TTL) value, set by its authoritative name server, that indicates how long the entry can be cached by other name servers in the system before being discarded. Therefore, caching of query results at name servers implies that name servers resolve address requests for both their authoritative and non-authoritative zone (the latter only until the TTL value expires). The most popular name server software is the Berkeley Internet Name Domain (BIND).

The DNS mechanism allows name servers to use two types of queries, that is iterative and recursive. When processing an iterative query, the name server returns either a final answer to the query from its local database (possibly cached data) or, if the entry is not available locally, a referral to another name server that may be able to answer it. (A referral includes the hostname of the name server and its corresponding IP address, that is both NS and A resource records.) If the query is recursive, the name server returns a final answer, querying in its turn any other name server necessary to resolve the name. Hence, the answer to a recursive query either contains the requested resolution or an error stating that a resolution for that domain name does not exist.

Most name servers within the DNS hierarchy send and accept only iterative queries. On the other hand, the operating system library routines (called resolvers) that handle queries for clients usually perform recursive name resolutions to the local name server. For example, on many Unix-based machines, gethostbyname() is the library routine that an application calls to issue a DNS query.
Figure 2.2: Basic steps for the name-to-address resolution of www.foo.com.

Figure 2.2 illustrates how a client typically finds the IP address corresponding to the hostname www.foo.com. First, the client application invokes a resolver that makes a recursive query to its local name server. The resolver knows the address of the local name server because it is configured in a system file (for fault tolerance reasons, some resolvers are configured with a list of servers). This request is typically recursive because the resolver cannot handle referrals. After making the request (step 1), the client waits for the local name server to iteratively try to resolve the name (www.foo.com in this example). If the local name server does not have a cached copy of the requested mapping, it sends an iterative query to a well-known root name server (currently, there are 13 root name servers in the world, the so-called .servers) to resolve the name (step 2). The root server returns a referral to a name server responsible for the top-level domain (com in this example) of the requested name (step 3). The local name server then iteratively queries this server (step 4). Since the subdomain foo.com has been delegated, the top-level server does not know the final answer, but responds with the IP address of the authoritative name server for the subdomain in which the host www.foo.com is located, i.e., ns.foo.com (step 5). The local name server uses this information to send the same query to the authoritative name server ns.foo.com (step 6). If the requested name exists, the local name server receives the IP address corresponding to www.foo.com and the associated TTL value denoting the period of validity of the name-to-address mapping (step 7). When the
local name server gets the answer for the authoritative name server, it saves the resource record (containing the name-to-address mapping and the TTL value) in its cache for subsequent requests. Finally, the name server returns the address to the client (step 8), that can now establish a TCP connection with the Web server (step 9).

In the example of Figure 2.2, we assume that there is only one intermediate name server between the root and the authoritative name server; in fact, they can be more or even less. We also assume that all queries are iterative; but combinations of recursive and iterative queries are also possible.

The address lookup phase shown in Figure 2.2 can require less steps if there is a valid entry for the name-to-address mapping in some caches, for example at the local name server (if the name has been previously resolved and the associated TTL is not timed out yet). The mapping can be also cached by the client browser. Indeed, address caching not only occurs at intermediate name servers, but also at the client browser level, as browsers typically cache the results of name resolutions, thus further reducing the number of DNS lookups. For example, Netscape Communicator caches name-to-address mappings for approximately 15 minutes.

So far, little research has been done about the impact of the DNS lookup phase on the overall cost to retrieve a Web object from a server. A surge of work has recently occurred in this field [47, 138, 155], mainly due to the increasing spread of geographically distributed Web-server systems and content distribution services (discussed in Chapter 4), that typically employ the DNS mechanism to redirect client requests to a nearby replicated server. A first finding is that the DNS lookup time is a significant fraction of the overall latency time when the name-to-address mapping is not cached, mainly because the authoritative name server can be distant from the local name server. In the next section, we will analyze the impact that the TTL value has on performance. Indeed, the probability that a DNS query will be resolved by a local name server is proportional to the TTL value. Since the local name server and the client are typically separated by no more than few routers (a recent study in [138] found that a typical client-name server distance is 8 hops), finding the address in the local name server's cache reduces substantially the address lookup time.

2.2.1 The Impact of TTL Values

The choice of the TTL value assigned by the authoritative name server determines a tradeoff among the load imposed on the authoritative name server for the RR
request, the responsiveness to changes in DNS information, and the DNS response
time performance.

Larger TTL values reduce the load on name servers as well as the network traffic, and augment the number of address lookups resolved locally. These factors improve the performance of the DNS mechanism, in terms of response time. On the other hand, large TTL values increase the staleness of the RR in caches of non authoritative name servers. So, local name servers may deliver stale information to their clients. This can happen, for example, when a machine with a particular hostname is moved from one location to another (thus having assigned a different IP address) and the resource record, corresponding to that hostname, changes in the authoritative name server. Due to the large TTL value associated with the old mapping, the update does not propagate quickly to other name servers.

Small TTL values allow a faster propagation of changes in the DNS database, but augment network traffic as more queries are sent to the authoritative name server. To reduce the impact that small TTL values have on network traffic, some name servers (hereafter, called *non-cooperative name servers*) enforce a minimum TTL on received RRs, even if the TTL value has been set to zero by the authoritative name server. This factor is important and we will return on it in Section 5.6.6.

Typical TTL values are in the order of a few hours or even a day. A recent study shows that the majority of authoritative name servers (about 80%) uses a default TTL value of 86400 seconds (or one day) for their domains [47].

### 2.3 HTTP Protocol

The HyperText Transfer Protocol (HTTP) is an application-layer protocol that defines how client browsers and Web servers interact. HTTP follows a simple request-response paradigm, where the server returns data to the client in response to a request. It has been designed on top of a connection-oriented protocol such as TCP and is referred to as a *stateless* protocol, because it does not preserve information across multiple HTTP connections.

Various versions of the HTTP protocol with different characteristics have been developed since the conception of the Web, starting with HTTP/0.9 to the latest HTTP/1.1. We survey the main differences of HTTP/1.1 from HTTP/1.0 that impact on the network and server delay components.

In version 1.0 of the HTTP protocol [28], a new TCP connection is used for each
request/reply exchange with the Web server. This means that the basic steps involved in the retrieval of an HTML object include the closing of the TCP/IP connection once the object has been received by the client. Each additional request for objects embedded within the Web page requires the establishment of a new TCP connection, as shown in Figure 2.3 for the embedded image. As regards the FIN packets flow in Figure 2.3, it is to be noted that in reality the client does not have to wait for the connection tear down to continue with the next request.

![Diagram of HTTP protocol flow](image)

**Figure 2.3**: Web transaction in HTTP/1.0.

HTTP/1.0 has several inherent performance inefficiencies both in the server and network usage, which have been addressed by a significant body of work in the literature [90, 113, 121, 124]. Since HTTP is typically implemented on top of TCP, the response latency is affected by the time needed to set up and tear down a TCP connection, that is each Web object retrieval suffers two round-trip times. The connection overhead contributes also to router congestion. Furthermore, since most Web objects are small, each transfer is affected by TCP slow start (the congestion control mechanism used in TCP during startup phases of the connection to increase TCP window size). On the server side, such a model makes inefficient use of server resources (in particular, CPU and memory), as the operating system of the Web server incurs a per connection overhead (each TCP connection must be managed, allocates send/receive buffers, and maintains state variables) and experiences TCP TIME_WAIT for each closed TCP connection.
The latest version of HTTP, that is HTTP/1.1, addresses these overheads by introducing persistent connections and pipelining [65]. Persistent connections last beyond a single request/response sequence, allowing a client to send several HTTP requests through the same TCP connection. Thus, the connection setup overhead is amortized and the impact of slow start is minimized. For example, embedded images in the base HTML file can be downloaded without new TCP connections, as shown in Figure 2.4.

![Diagram of Web transaction in HTTP/1.1 with persistent connections.](image1)

![Diagram of Web transaction in HTTP/1.1 with persistent connections and pipelining.](image2)

The pipelining technique allows to send multiple requests to a server over the same connection without having to wait for any response to previous ones. The server sends the responses in the order of the received requests. An example of the corresponding client/server interaction is shown in Figure 2.5, in which the client, after having parsed the base HTML file, issues new requests for the embedded images (multiple client requests can be also carried by the same TCP segment). Pipelining has the effect of reducing the number of requests and responses, thus greatly improving performance [121]. A quantitative analysis of the response time for a typical Web page under different versions of the HTTP protocol is provided in [97].

Persistent connections and pipelining have been proved to reduce significantly client latency and network traffic [75, 90, 92, 121]. Still, they do not eliminate all
overheads of TCP. Indeed, they may even introduce new performance penalties, especially when the bottleneck is at the server [24].

The limitation of pipelining is that servers must send responses in the same order of the requests in the pipeline. This constraint causes delays when a slow response holds up all other responses in the pipeline.

Persistent connections increase the number and length of open connections at the server, which can have a significant negative impact on server throughput, especially in the case of busy servers [24, 113]. A larger number of open connections increases both the memory pressure and the context-switching overhead in the server system. To limit the number of open connections, servers close those connections that remain idle for a persistent connection timeout period (e.g., 15 seconds is the default timeout value in the Apache Web server). It should be noted that when the objects embedded within a composite page (e.g., images or advertisements) are stored on another server different from the previously contacted one, the HTTP/1.1 support for persistent connections does not eliminate the TCP connection delay. Moreover, access to objects located on a different server often requires a new address resolution to obtain the IP address corresponding to the name of the other server.

To speed up the retrieval of content and improve user-perceived performance, many popular Web browsers often open multiple parallel TCP connections (typically four) to the same server, so to fetch the various objects that make up a Web page [4]. An effect of this implementation is that the client is able to quickly start the display of multiple objects. However, such a behavior may increase the number of open connections and degrade the throughput of a busy server.

## 2.4 Web Servers

A Web server is a combination of a hardware platform, operating system, networking software, contents, and a HTTP server. Basically, the Web server listens to a well-known port (typically, port 80) for HTTP requests coming from clients over the network. The server establishes the requested connection with the client, reads the HTTP request from the client, sends back the requested object, and returns to its listening function.

To speed up the service, Web servers handle more than one request at a time through concurrency strategies. Some alternative architectures to achieve concurrency are described below.
Multi-process architecture. A copy of the HTTP process is assigned to each connection in order to execute sequentially the basic steps that make up the service of a connection. For example, for a static content request on HTTP/1.0, the steps include: (1) accepting the client connection, (2) reading the request, (3) finding the requested content file, (4) sending the response header to the client, (5) reading the file data, (6) and sending the requested content to the client. Since multiple processes are employed, many connections can be served concurrently. In order to reduce forking overhead, a set of persistent processes is pre-forked during Web server initialization. The drawbacks of multi-process architectures are the content switching and IPC overheads. An example is Apache 1.3 [9] on Unix operating systems.

Multi-threaded architecture. In multi-threaded servers, multiple concurrent threads are executed within a single address space. Each thread performs all the steps needed to fulfill the connection, similarly to the use of a process in the multi-process architecture. A group of threads is usually pre-spawned during server initialization. When a new connection arrives, the operating system selects a thread from the pool to accept it. The multi-threaded architecture requires that the operating system underlying the Web server supports kernel threads. Examples of such architecture are provided by Apache 1.3 on Microsoft Windows NT, Microsoft's Internet Information Server (IIS), and the forthcoming Apache 2.0 on Unix systems.

Single-process event-driven architecture. Event-driven servers allow a single process to perform concurrent processing for all connections being handled by the server. Their drawback is that the delay in handling requests for one connection can affect the response time of other connections. Examples of such architecture are provided by the Zeus Web server [161] and the Flash Web server [126].

Many Web servers are available that employ various performance optimizations (depending also on the operating system platform on which the servers run) and thus have different performance characteristics [126]. Apache is a freely available Web server [9] and is currently the most deployed Web server in Internet. According to recent estimates, it is used by over 60% of Web sites [120]. Apache version 1.3 is a multiprocess-based server, implemented in user space, with pre-forking of several processes. When a new connection arrives, the operating system delivers it to an available child process that is waiting on the listen socket.
Section 2.4. Web Servers

The rest of this section is outlined as follows. Section 2.4.1 describes the diverse types of content provided by Web servers. Section 2.4.2 outlines multi-layer Web server architectures.

2.4.1 Content Provided by Web Servers

The information serviced at a Web site can be classified into four principal types. The first is static content, where data remain unchanged over long periods of time (e.g., gif and jpg images). Web servers typically obtain the static content from corresponding files stored at the local file system.

The second is dynamic content, where the response is generated dynamically by an auxiliary application program at the time a request is made (e.g., a search query in a catalog). The information is created on user demand (i.e., the information is personalized) and typically depends on data provided by the user. Several technologies are now available that support dynamic content generation. They are briefly reviewed in the next section.

The third type is volatile content, where the information provided by the Web server is the same for all client requests, even if the content changes frequently (e.g., news, weather reports). Volatile content differs from static content and from dynamic content, as it is created and modified when some event occurs (e.g., an update in the weather forecasts) and the same information is provided to all users accessing the site at the same time.

Finally, the fourth type is secure content, where static or dynamic information between the server and the authenticated client travels encrypted by means of the Secure Sockets Layer (SSL) protocol [10, 106].

The impact of the content type on Web server performance varies greatly, because of the diverse server resources demands required. While the service of static content requires only a file fetch (whose resources consumption mainly depends on the file size), dynamic content processing may involve more expensive CPU computation, as well as database searching. Thus, the time to serve a request for dynamic content is often significantly higher than that required to satisfy a request for static content. It is not uncommon that the generation of a dynamic page (which includes dynamic content as well as static files containing images or other multimedia data) consumes over a second, while the average CPU time to satisfy a static request is on the order of few milliseconds. Secure content also imposes a significant overhead (by up to two orders of magnitude with respect to normal pages [10]), as it requires CPU intensive
computation for data encryption.

2.4.2 Multi-tier Architectures

Web sites devoted to electronic commerce are usually structured in layers rather than in a flat architecture in order to improve system functionality, performance, scalability, and reliability [106]. The architecture of E-business sites typically consists of three layers, as shown in Figure 2.6.

![Multi-tier Architecture Diagram](image)

**Figure 2.6: Multi-tier architecture.**

The *application server* is the software that handles all application operations between the client and the site’s back-end databases. In general, the application server receives requests for dynamic content from the Web server (e.g., a search in a catalog) and interacts with the *database server*, that provides access to one or more shared databases. When the application is completed, the result is sent back to the Web server that replies to the client. Application servers can be implemented using different technologies that allow to create dynamic Web pages.

The Common Gateway Interface (CGI) provides an interface to run CGI scripts (e.g., programs written in Perl or C) outside the Web server and to return data to the Web server that forwards the content to the client. The CGI interface requires that the server initializes an environment containing the script parameters, and then forks a process to execute the script. Drawbacks in using CGI scripts are the overhead to fork a new process that handles each CGI request, and the complexity of building applications that require several interactions with the user (e.g., cookies have to be used to maintain the application state from one interaction to the next one). CGI scripts are still widely used but, because of their high overhead, they are being replaced by more efficient approaches, like the FastCGI interface. This is an extension of
Section 2.5. Performance Metrics for Web Services

CGI that removes the need for process creation for each request by making the CGI processes persistent.

Another approach for dynamic content generation is that of an application running within the context of the Web server: the application is invoked through some Application Program Interface (API) provided by the Web server software (e.g., Microsoft’s Internet Server API or Netscape’s Netscape Server API). APIs allow to perform server-side processing without forking a new process by dynamically loading precompiled code. Other solutions use server-side scripting languages, such as Hypertext Preprocessor (PHP) or Microsoft’s Active Server Pages (ASP). The script, which is embedded within the HTML page, is interpreted and the operations in it are executed. Performance of dynamic Web page generation technologies are analyzed in [88].

![Diagram](image.png)

Figure 2.7: Web transaction in a 3-tier architecture.

An example of Web transaction on a Web site with a multi-tier architecture is shown in Figure 2.7. The client sends a request to the Web server, which sends a request to an application server, which in its turn asks data from the database server. The application server receives the data from the database server and builds an HTML page that is returned to the Web server. This latter replies to the client.

2.5 Performance Metrics for Web Services

Web performance can be analyzed from different points of view. For instance, user-perceived Web performance has to do with fast response time and no connections
refused, while users do not know neither care of complexity of Web infrastructure and technology. On the other hand, a Web site administrator is mainly interested in Web server throughput, while e-commerce retailers can be interested in the amount of money earned from sales. In this section, we discuss the main performance metrics for Web services that can be measured at the server side as well as at the client side.

2.5.1 **Server-side Performance Metrics**

Latency time and throughput are the most important key performance metrics from the Web server perspective.

*Server throughput* \( X \) is defined as the number of requests that the Web server, considered as a black box, completes per time unit, that is \( X = N/T \), where \( N \) is the number of requests completed during the measurement interval \( T \).

Different indexes can be used to measure the throughput. The most popular is *HTTP requests per second* (or hits per second), that is the number of requests for Web objects served in each second. This index is also usually referred to as *connection per second*. It is to be noted that throughput as measured in connections per second is the same as requests per second for HTTP/1.0, while the two measures differ when HTTP/1.1 with persistent connections is employed in the client/server interaction.

Due to the large variability in the size of Web objects [13, 51, 130], the throughput is also measured in terms of Mbytes per second, where \( N \) in \( X = N/T \) represents the total size in bytes of the transferred files during the interval \( T \). This index allows to measure the network bandwidth demand of the Web server.

*Server latency*, also referred to as *server delay* or *server residence time*, measures the time required to complete a request at the server side that does not include any Internet delays. On the other hand, latency is equal to the sum of the waiting time plus the service time over all server resources, both hardware (e.g., CPU and I/O devices) and software (e.g., an HTTP process). Server throughput and latency are related through Little’s Law.

In highly variable systems subject to unpredictable traffic spikes as Web servers are, percentiles or the cumulative distribution of latency time are more significant metrics than average values. Indeed, service level agreements (SLAs) in terms of performance are usually measured as the \( K \)-percentile of the server delay that must be less than \( Y \) seconds [93]. Typical values for \( K \) are 90- or 95- percentile.

While throughput and latency are usually reported as average measures during the Web server operation, Web site administrators should also know the upper bound on
the sustainable server throughput (that is, the maximum throughput the Web server is capable of processing), as well as the lower bound on the latency (that is, the minimum latency the server can achieve). These optimistic performance bounds, as well as their pessimistic counterparts (that is, the lower bound on the server throughput and the upper bound on the latency) are necessary to define appropriate SLAs. Furthermore, the maximum server throughput (measured in terms of Mb/s) has to be compared with the network bandwidth of the server's link to the Internet backbone in order to identify if the latter is the bottleneck.

Since content provided by Web servers impose different server resources demands, as pointed out in Section 2.4.1, server-side metrics should be differentiated according to the content class. For instance, the server throughput for static files can be of several hundred requests per second, while it is reduced by a factor of about ten for dynamic pages (or even more when interactions with database servers are required) [84]. As an example, the TPC-W benchmark [149], that has been defined by the Transaction Processing Council (TPC) to evaluate the performance of e-business sites, has three types of throughput metrics depending on the level of user interaction with the site, that is browsing, shopping, and ordering.

### 2.5.2 Client-side Performance Metrics

The crucial performance metric for users is the response time, as it directly correlates with a user's perception of the quality of service. It is typically measured for an entire page, because this entity is perceived by the user as a single document. The *page response time* corresponds to the interval elapsed between the submission of the client request for a given page and the arrival at the client of all objects corresponding to the page request. It includes DNS lookup delay, TCP connection overheads, delays at Web server, and network transmission time. Recent analysis found that users are not willing to tolerate response times greater than eight, ten seconds [31]. Furthermore, their tolerance for latency decreases over the duration of interaction with a site and a one-second improvement in response time reduces the rate of abandonment of the Web site from 30% to 6%.

From the perspective of the client, it is hard to differentiate between server and network delays. A good approximation, especially for small transfers, is to estimate the server delay component to the overall object retrieval latency as the time that elapses between the sending of the HTTP request and the receipt of the first bytes of response [78]. However, for large data transfers with a busy server, there can be
significant server delays after this point of the transaction [25].

As a consequence of the substantial changes transforming the Web from a communication and browsing infrastructure to a medium for conducting personal businesses and e-commerce, demands placed on Web services continue to grow and Web servers are becoming more stressed than ever.

Because of the complexity of the Web infrastructure, performance problems may arise in many points during a Web transaction, as outlined in Section 2.1. For instance, they may occur at network level because of congested Internet backbones, as well as at server level either because of underprovisioned capacity or an 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 related research. In this thesis, we are mainly interested in reducing the Web server delay component, which is the component under the control of Web site's designers. Indeed, Web performance perceived by end users is increasingly dominated by server delays, especially when contacting busy servers [25]. Recent measures suggest that the Web servers contribute about 40% of the delay in a Web transaction [78] and it is likely that this percentage will augment in the near future. However, 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 [71]. Other network improvements, such as private peering agreements between backbone providers, the rapid adoption of ISDN network, DSL lines, and cable modems contribute to reduce network latency. With the network bandwidth increasing much faster than the server capacity (e.g., the deployment of Gigabit wide-area networks), the bottleneck will be more on the server side. In this scenario, the brute-force approach of adding ever increasing network and server capacities will not solve the Web scalability problem in a foreseeable future.

Many efforts have been directed at measuring the behavior of single Web servers under overload and improving their performance either at the level of the operating system or the server application itself [21, 20, 76, 119, 126]. Pai et al. [126] have designed the Flash Web server, which, using an asymmetric multiprocess event-driven architecture, ensures that its threads and processes are never blocked. Nahum et al. [119] have analyzed how the server general-purpose operating system and the network protocol stack can be improved to provide support for high-performing Web servers. Hu et al. [76] have proposed some techniques to improve the performance of
the Apache Web server.

While those efforts focus on a single-server platform, in this thesis we consider scalable Web-server systems composed by multiple nodes, which are the the only viable and cost-effective solution to provide Web services that keep up with the increasing performance demands.

### 2.6 Multiple Web Servers

Web site administrators constantly face the need to increase server capacity. The first option used to scale Web services is to upgrade the Web server to a larger, faster machine. This strategy, referred to as **scale-up** [56], simply consists in expanding a system by incrementally adding more resources (e.g., CPUs, disks, network interfaces) to an existing node. While scale-up relieves short-term pressure, it is neither a cost-effective nor a long-term solution, considering the steep growth in the client demand curve which characterizes the Web (the number of users online is growing at 91% per annum).

To handle significant traffic, Web sites must use multiple servers running on different machines. A simple approach is to replicate the content on a **mirrored-server** architecture. This load sharing technique lets users manually select alternative names for a Web site; as a consequence, any user can send Web requests to the node of its choice, for example to the node that is geographically closest. The obvious advantage is the simplicity of implementation. However, this architecture has several disadvantages, as it is not user-transparent, nor does it allows to control the request distribution, leading to poor load sharing because users do not usually know about load of the server nodes and network conditions. Furthermore, the visibility of individual nodes can also potentially raise security concerns and it is very difficult to guarantee consistency of replicated information.

A more appealing solution to keep up with ever increasing request load is to deploy a scalable, distributed architecture that can route incoming requests transparently among several server nodes. A **scalable Web-server system** is any architecture consisting of multiple Web-server nodes, distributed on LANs or WANs (namely, locally and geographically Web-server systems), with a mechanism to spread incoming client requests among the nodes. The approach in which the system capabilities are expanded by adding more nodes, complete with processors, storage, and bandwidths, is typically referred to as **scale-out** [56]. We further distinguish between **local scale-out**,
when the collection of nodes resides typically at a single location and *global scale-out*, when the nodes are located at different geographical locations. Figure 2.8 summarizes the different approaches to achieve system scalability.

![Diagram](image)

**Figure 2.8: Scalability in Web-server systems.**

Information is typically distributed among the server nodes of the Web system in two ways: content tree replication across independent file systems running on the servers, or information sharing by means of a distributed file system such as Andrew File System or Distributed File System. Besides the storage overhead, content replication requires to propagate to all the nodes in a short period of time a content update. On the other hand, a distributed file system increases the load on the server nodes as they have first to obtain the file from the file server before sending it to the client. Furthermore, a distributed file system solution works fine only for local Web-server systems.

A scalable Web-server system 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. In this thesis, we only consider user-transparent solutions. Thus, users interact with the Web-server system as if it were a single high performance server. On the other hand, the client may be aware of the effects of some dispatching mechanisms. However, it is not affected by the interaction with the Web system and does not need modification.
2.6.1 Attributes for Web-server Systems

Besides architecture transparency, in this thesis we consider scalable Web-server system that address the following attributes.

**Performance.** Performance must be guaranteed because it is a critical measure for the success of any Web site. Web systems have to face two main challenges: maximizing server responsiveness and minimizing network costs. The first is addressed by locally distributed architectures, while the latter can be achieved through geographically distributed architectures.

**Scalability.** The growth of Web traffic, which continues with no near-term end in sight, makes increasingly difficult to design a Web-server architecture capable of meeting increasing user demand. Furthermore, sudden spikes in traffic increase the difficulty to size server capacity. So, a system architecture that scales out (by adding more nodes) incrementally with demand becomes a necessary feature to provide a long-term solution. Scalable services must grow without hurting performance or availability. Scalable services use up-to-date content, are distributed and redundant, inherently dynamic and self-healing, proactive in their distribution of data when necessary.

In this thesis, Web system scalability is defined as the ability to support large numbers of accesses and resources while still providing adequate performance.

**Load sharing.** Site load must be evenly distributed among the server nodes belonging to the system, so as to reduce user-perceived latency time and to achieve the highest performance. Load sharing is instrumental in obtaining high performance server systems. It can be achieved by several methods, with diverse degrees of effectiveness. We will propose and compare load sharing mechanisms and algorithms in Chapters 5, 6, and 7.

**Backword compatibility.** A scalable Web-server system should operate transparently with the existing Web, i.e., it should be readily deployable over currently existing Internet infrastructure. So the mechanism to distribute the requests and any other adopted solution should work with actual clients and name servers, without requiring modifications to existing protocols or client code. To this purpose, we define *client compatible* a solution that does not require any modification on the client, while *server compatible* a solution that does not require
modifications on the operating system of the servers. The two requirements have different impacts, that is client compatibility is much more important than server compatibility. While the violation of server compatibility entails the modification of the Web servers which are under control of the Web site technical management, solutions which are not client compatible require modifications to components of the Web infrastructure which are not controlled by the content providers.

**Support to Internet applications.** The solution chosen for spreading the requests across the multiple nodes should support other applications in addition to Web services. Architectures that are independent of the particular Internet service are preferable. The ideal solution would be one that offers support for any IP, TCP, or UDP protocol.

Other attributes of Web-server systems that will not be pursued in this thesis include reliability, security, and content accessibility. The description of some Web-server systems that address some of these issues can be found in [114, 158].

### 2.6.2 Routing Along the Web Request Path

The main components of a typical multi-node Web system include a *routing mechanism* to direct the client request to the target server node, a *dispatching algorithm* to select the Web server node best suited to respond, and an *executor* to carry out the dispatching algorithms and support the routing mechanism. In this section, we examine where the decision on client request routing can occur along the request path from the client to the Web site.

Through the analysis of a Web transaction discussed in Section 2.1, we identify four levels at which the request routing can be deployed:

- at the *Web client* level: the entity responsible for the request assignment is the Web client that originates the request;

- at the *DNS* level, during the address resolution phase: the entity in charge of the request routing is the authoritative DNS server for the Web site;

- at the *network* level: this approach can be carried either by the network itself (the so-called active networks [147]), where intermediate routers transparently
redirect packets to a target server node, or by Web proxy servers located along
the path between the client and the server system.

- at the Web system level: the entity in charge for the request assignment is the
  Web server or some dispatching entity in front of it.

2.6.3 Solutions not Examined

In this thesis, we basically consider all approaches that are based on the content
provider intervention. Our basic premise is the compatibility of all proposed solu-
tions with existing Web standards and protocols so that any considered architecture,
algorithm and mechanism could be immediately adopted without any limiting as-
sumption. Moreover, we consider architectures for Web sites that use a single name
to make the distributed nature of the service transparent to the users. Therefore, we
will focus on solutions that occur either at DNS level, at Web system level, or at a
combination of both. Client-based selection will not be investigated any further in
this thesis as such an approach is not universally applicable; indeed, routing at the
client is not always client compatible [36]. Also, we do not consider either network
solutions based on proxy caching (e.g., reverse proxy caching) or outsourcing solutions
such as Web hosting and co-location where the content provider delegates all services
to a Web hosting company (whose Web-server system stores and provides access to
multiple Web sites).

To summarize, we will not investigate the following techniques and solutions, for
which we provide some references where additional reading can be found:

- client-side solutions [19, 116, 150, 159];
- Web server caching techniques [26, 151];
- outsourcing solutions such as Web site hosting and co-location [5, 15, 43, 102].

In the next two chapters, we will classify and describe some techniques to split
the traffic load among the server nodes, discussing both the alternative architectures
and the load sharing policies for scalable Web systems. First, in Chapter 3 we will
consider locally distributed Web systems, where the server nodes reside at a single
location. Then, in Chapter 4 we will discuss architectures and load sharing policies
for geographically distributed Web systems.
Chapter 3

Locally Distributed Web-server Systems

This chapter presents and discusses the various approaches for managing *locally distributed Web systems*. We describe architectures, routing mechanisms, and dispatching algorithms to design local Web-server systems and identify some of the issues associated with setting up and managing such systems for highly accessed Web sites. We examine how local architectures and related management algorithms satisfy the scalability and performance requirements of Web services. We analyze the efficiency and the limitations of the different approaches and the tradeoff among the alternatives with the aim to identify the characteristics of each approach and their effectiveness on Web performance.

The rest of this chapter is organized as follows. Section 3.1 discusses and classifies locally distributed architectures. The main target architectures of this chapter are tightly coupled ones with a single system interface, that we call *Web-server clusters* or simply *Web clusters*, in which a front-end node, called *Web switch*, distributes all client requests to the server nodes. Section 3.2 covers the routing mechanisms that characterize Web cluster alternatives. Section 3.3 is devoted to consideration of options in load sharing algorithms for the class of local Web-server systems. We present a taxonomy of the policies that have been developed in recent years by focusing on the issues that each policy addresses. In Section 3.4, we classify various academic and commercial products according to the routing mechanism that the Web switch uses to distribute the requests among the servers. Section 3.5 presents some possible extensions to the basic system architecture. Section 3.6 concludes the chapter with some remarks on open research issues.
3.1 System Architectures

A locally distributed Web-server system is composed by a tightly coupled architecture placed at a single location. The nodes are networked to each other through a LAN. A high-level view of a locally distributed architecture is shown in Figure 3.1, having omitted how the request routing occurs in the system; the various methods will be described in the next sections.

![Diagram of Web-server system]

Figure 3.1: A locally distributed Web-server system.

In this thesis, we consider only distributed architectures which adhere to the transparency requirement by providing a single virtual interface, at least at the site hostname level, to the outside world. Distributed architectures can be differentiated depending on the name virtualization being extended at IP level or not. Given a set of server nodes that host a Web site at a single location, there are two main approaches:

- the server nodes make their IP addresses visible to clients: it implies that a client application may be aware of the multiple server nodes because their IP addresses are not kept hidden;

- the server nodes mask their IP addresses to clients: the only visible address is
Section 3.1. System Architectures

a Virtual IP (VIP) address corresponding to one device located in front of the set of servers.

We call the former solution Distributed Web-server System (DWS), and the latter solution Web-server Cluster (WSC). Each solution requires different mechanisms for distributing the load. The former solution is older and typically carries out the request routing at DNS level (or some other centralized entity); the latter is more recent and carries out the request routing only at the Web system level. We can anticipate that, in a local environment, a Web-server cluster architecture is preferable for many reasons. Thus, in this chapter we describe WSC architectures, while we postpone DWS solutions, which are more appropriate to a geographical context, to the next chapter.

3.1.1 Web-server Clusters

A Web-server cluster (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 cluster node 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 (e.g., SP-2) where each node satisfies all previous characteristics can be assimilated to a Web-server cluster.

Although a Web cluster may consist of tens of nodes, it is publicized with one hostname (e.g., www.foo.com) and one virtual IP address (e.g., 144.55.62.18). Thus, the authoritative DNS server for the Web site performs a one-to-one mapping by translating the site hostname into the VIP address. The latter corresponds to the IP address of a dedicated front-end node(s) that interfaces the rest of the cluster nodes with the Internet, thus making the distribute nature of the cluster completely transparent to the clients (i.e., the clients are unaware of the existence of the cluster). The front-end node, hereafter called Web switch, acts as a centralized dispatcher with fine-grained control on client requests assignment. The switch receives inbound packets from clients destined to the VIP address and routes them to a target server node selected through a dispatching algorithm.

A high-level view of a basic Web cluster comprising the switch and N servers is shown in Figure 3.2. It is to be noted that the response line does not appear as there are a few alternatives for it that will be outlined in Section 3.2. Also, for the
sake of clarity, only Web-server nodes are shown; however, back-end nodes that act as application and/or database servers can be integrated in the system, as it will be seen in Section 3.5.2.

![Diagram of Web-server cluster]

**Figure 3.2: Architecture of a Web-server cluster.**

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 private address may correspond to either an IP address or a lower-layer (MAC) address. As explained in the next section, there are various techniques to deploy Web clusters; we will differentiate them through the mechanism being used to translate the VIP address of the packets that reach the switch into the private address of the selected server node.

Implementations of the Web switch can be based on either special-purpose hardware devices plugged into the network or software modules running on a common operating system. In this chapter, we use the term Web switch to refer to the dispatching entity in general. The term does not imply that the Web switch is a hardware device, nor it has to be understood as the entity that forwards either frames based on link layer addresses or packets based on layer-3 and layer-4 information.

To conclude this section, we review some alternative terminology used in literature
to describe Web clustering techniques. Web clusters are also referred to as Web farms, meaning the collection of all the servers, applications, and data at a particular site [56]. We prefer the term Web cluster, as Web farm is also used to denote Web site hosting on multiple servers (i.e., co-location).

Others refer to a Web switch as a server load balancing product. However, this term encompasses a wide range of technologies; moreover, deployed Web cluster solutions provide load sharing rather that load balancing [79, 137]. Load balancing strives to equalize the servers load, while load sharing attempts to smooth out transient peak overload periods on some nodes [60].

3.2 Routing Mechanisms for Web-server Clusters

The Web switch plays a key role in a Web cluster. For that reason, we first broadly classify the architecture alternatives according to the OSI protocol stack layer at which the Web switch operates the request routing, that is layer-4 or layer-7 Web switches.

The main difference in Web cluster architectures derives from the kind of information available to the Web switch to perform routing decision.

- Layer-4 Web switches perform content-blind routing (also referred to as immediate binding), because they determine the target server when the client establishes the TCP/IP connection, upon the arrival of the TCP SYN packet.

- Layer-7 Web switches can execute content-aware routing (also referred to as delayed binding), by letting the switch establish a complete TCP connection with the client, examine the HTTP request and then relay it to the target server.

It is to be pointed out that we refer to layer-7 Web switches according to the ISO/OSI protocol layers, where the application layer is the seventh. Others refer to switches that perform content-aware routing as layer-5 Web switches.

Both 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 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 or return its response to the Web switch, that in its turn sends the response back to the client. We refer to the architectures deploying the first type
of data flow as one-way architectures, while those implementing the latter are called two-way architectures. Figure 3.3 summarizes the taxonomy for Web clusters that we have examined so far.

![Taxonomy of Web cluster architectures.](image)

**Figure 3.3: Taxonomy of Web cluster architectures.**

### 3.2.1 Layer-4 Solutions

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 switch examines its header and determines, on the basis of the bits in 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 target server. If the inbound packet is not for a new connection, the Web switch looks up the binding table to ascertain whether the packet belongs to an existing connection. If it does, the Web switch determines through the binding table which server is in charge for the connection and routes the packet to it. In the event that the packet does not correspond to an established connection, the Web switch drops the packet.
Section 3.2. Routing Mechanisms for Web-server Clusters

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

Layer-4 Web clusters can be classified on the basis of both the mechanism used by the Web switch to route inbound packets to the target server and the packet way between the client and server. The main difference is in the return way that is, server-to-client. In two-way architectures both inbound and outbound packets are rewritten at TCP/IP level by the Web switch, while in one-way architectures only inbound packets flow through the Web switch. The routing to the target server can be accomplished by either rewriting the IP destination address, encapsulating the IP datagram within another IP datagram or forwarding the packet at MAC level.

Two-way (through the Web Switch)

In two-way architectures, each server in the cluster is configured with a unique IP address (i.e., 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 3.4.

![Layer-4 two-way architecture.](image)

Figure 3.4: Layer-4 two-way architecture.
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 them is the address of the server that served the request. Thus, 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 checksum for both packet flows.

One-way (through the Web Switch)

In one-way architectures only inbound packets flow through the Web switch, thus allowing a separate high-bandwidth network connection for outbound packets. Figure 3.5 shows the packets flow; the level of the private address is intentionally not specified as it can be either at IP level (layer-3) or MAC level (layer-2).

![Figure 3.5: Layer-4 one-way architecture.](image)

The routing to the target server can be accomplished by either rewriting the IP destination address and recalculating the TCP/IP checksum of the inbound packet (i.e., packet rewriting), by encapsulating each packet within another packet (i.e., packet
Section 3.2.  Routing Mechanisms for Web-server Clusters

* tunneling) or by forwarding the packet at MAC level (i.e., packet forwarding).

We describe how each approach works in detail below.

**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 server private addresses are at IP level. The difference from two-way architectures is in the modification of the source address of outbound packets. Indeed, the Web server, before sending directly the response packets to the client, replaces its IP address with the VIP address and recalculates the IP and TCP header checksum. To differentiate the packet rewriting performed in two-way architectures by that of one-way ones, we call the first *double-rewriting*, while the latter *single-rewriting*.

**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 [129]. 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, whose header contains the VIP address and the server IP address as source and destination address, respectively. As in packet rewriting, the server private addresses are at IP level.

To have this approach work, the servers must support IP tunneling and all have one of their tunnel device configured with the VIP address. Upon receiving the encapsulated packet, the target server strips the IP header off and finds that the inside packet is destined for the VIP address configured on its tunnel device. Then, the server processes the request and returns the response directly to the client.

**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 segment of LAN.

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. That is, each server is configured with the VIP address as secondary address. This may be
done through the use of loopback interface aliasing (e.g., 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, it would be a collision). Hence, the Web switch can forward the inbound packet to the target server 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. Therefore, packet forwarding is also referred to as MAC address translation. Unlike previous approaches, the server private addresses are 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.

### 3.2.2 Layer-7 Solutions

Layer-7 Web switches work at application level, thus allowing content-based request distribution. The mechanisms for content-aware routing are more complex than those for content-blind routing. This is because the HTTP request is first inspected before a decision is made about which server node should handle the request. In order to determine the request content, the Web switch must first establish a TCP connection with the client (i.e., the three-way handshake for the TCP connection establishment phase must be completed between the client and the Web switch) and then receive the HTTP request (i.e., the application level information). 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 3.6 shows the different instant in which the request decision is made by a Web switch that deploys content-blind or content-aware routing in the case of a new connection.

Unlike layer-4 Web switches where the Web content is replicated on each server node or shared through a distributed file system, layer-7 Web switches allow also content/type partition on Web server nodes specialized for certain types of requests.

Similarly to layer-4 solutions, layer-7 Web switch architectures can be classified on the basis of the mechanism used by the switch to redirect inbound packets to the target server and the way back of packets from server to client. The main classification
is based on the data flow through the Web switch, discriminating between one-way architectures and two-way architectures.

**Two-way (through the Web Switch)**

In two-way architectures outbound traffic must pass back through the switch, as depicted in Figure 3.7.

The proposed approaches differ from each other in the way data are routed from the Web switch to the target server.

**TCP gateway.** An application level proxy running on the switch mediates the communication between the client and the server. The proxy accepts client connections and maintains 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 is an optimization of the TCP gateway approach, in that data forwarding occurs at network level (between the network interface driver and the TCP/IP stack) and is done directly by the operating system.

Once the TCP connection between the client and the Web switch is established and the persistent TCP connection between the switch and the target server has been chosen, the two connections are spliced together; so IP packets are forwarded from one endpoint to the other, without having to go across the TCP
layer to the application layer on the switch. Thus, as the client-to-server binding is determined, the switch handles the consequent 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.

While the one we have illustrated is a software-based switch, hardware-based switches can be also designed as described in [10].

**One-way (through the Web Switch)**

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

**TCP handoff.** Once it has established the TCP connection with the client and selected the target server, the Web switch hands off its endpoint of the TCP connection to the server [125]. The handoff protocol is layered on top of TCP and runs on the Web switch and the servers, thus requiring changes in their operating systems. The handoff mechanism remains transparent to the client, as data sent by the servers appear to be coming from the Web switch and 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. The handoff mechanism allows also to handle HTTP persistent connections by letting the Web switch assign HTTP requests in the same connection to different target servers [16]. To support persistent connections, either the handoff protocol can be extended by allowing the Web switch to migrate a connection between servers (multiple handoff) or the first target server forwards the request it cannot serve to a second server (always selected by the switch), which in its turn sends the response back to the client (back-end forwarding).

TCP connection hop. This a software-based proprietary solution developed by Resonate [135]. 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 [135]. Since the server shares the same VIP address, it can reply directly to the client. Acknowledgment packets and persistent session information from clients are then managed by the switch.
3.2.3 Routing Mechanisms Comparison

Figure 3.9 summarizes the taxonomy for Web clusters routing mechanisms that we have discussed in this section; it details further the taxonomy previously depicted in Figure 3.3.

![Flowchart showing Web cluster architecture]

Figure 3.9: Detailed taxonomy of Web cluster architectures.

In the rest of this section, we will first compare the different approaches we have identified inside content-blind and content-aware routing. Then, we will confront the two overall mechanisms.

Content-blind Routing

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. Their problem is that the Web switch must rewrite inbound as well as outbound packets, and outbound packets typically outnumber incoming 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 dedicated hardware support is provided for the latter operation.

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, as the outbound packets do not need to traverse the switch. 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. As to the different approaches classified within the one-way solution, packet
single-rewriting sustains the same overhead as rewriting in both directions but it reduces switch bottlenecks because the more numerous server-to-client packets are rewritten by the Web server and not by the Web switch. However, to accomplish this rewriting task, packet single-rewriting requires to modify the kernel of the server operating system.

Packet forwarding mechanisms are an effort to resolve the overhead of packet rewriting. As the Web switch processes only inbound packets and, moreover, at MAC level (thus avoiding 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 in practice little impact since the Web switch and the servers are likely to be connected by a single LAN.

Solutions that employ packet tunneling have good scalability (although lower than packet forwarding). However, they require server support for IP tunneling, which is not yet a standard for operating systems.

**Content-aware Routing**

In two-way solutions, caching can be implemented on the Web switch, thus allowing it to reply directly to a request if it can be satisfied from the cache with a consequent load decrease on server nodes. The main advantage of the TCP-gateway approach is its simplicity that allows to implement it using standard operating systems. However, the Web switch can easily become a bottleneck as both request and response data flows through it up to the application level. TCP splicing reduces TCP gateway overhead, as 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 become the bottleneck of the cluster as it has to modify the TCP/IP headers.

Layer-7 Web switches that use a one-way solution 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 [17]. The main disadvantage of one-way solutions lies in that they require modifying the operating system of both the Web switch and the servers.
Content-blind vs. Content-aware Routing

Content-aware routing takes the request content into account when selecting the target server. Potential advantages of layer-7 Web switches over layer-4 Web ones include:

- increased performance due to higher cache hit rates;
- the capability to employ specialized Web server nodes (e.g., streaming content, dynamic content);
- the capability to partition the Web content among the servers, thus increasing secondary storage scalability;
- the capability to assign subsequent SSL sessions to the same server (this feature is also known as sticky or persistent session support). 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 3.3.3). Indeed, stateful services can be identified through the service port (e.g., 443 for SSL) and a connection reuse timeout can be set [79].

Moreover, content-aware routing allows a finer-grained assignment control than content-blind routing when HTTP/1.1 persistent connections are employed in the client/cluster interaction. Indeed, 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 assigns 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 TCP connection level. With HTTP/1.0, there is no difference between content-blind and content-aware routing as to the granularity control, because a one-to-one correspondence exists between HTTP request and TCP connection.

Still, content-aware routing introduces an additional processing overhead at the dispatching entity and may cause the Web switch to become the system bottleneck, thus limiting seriously cluster scalability [17, 143]. As an example, Aron et al. show
Table 3.1: A summary of local routing mechanisms.

in [17] 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 using the same hardware is able to sustain a throughput up to 20000 conn/sec. To overcome this drawback, alternative designs for high performance Web server systems, which combine content-blind and content-aware request distribution, have been proposed and are described later in Section 3.5.

Table 3.1 outlines the features of the various approaches we have discussed and their tradeoffs.

### 3.3 Dispatching Algorithms for Web-server Clusters

In this section, we describe policies applicable for distributing requests in Web-server clusters. The dispatching policy has an immediate effect on both performance and scalability of the system. If incoming requests are distributed among the server nodes in an adequate fashion, the overall scalability and performance are improved.

The Web switch may use various global scheduling policies\(^1\) to assign the load to the nodes of a Web cluster. Global scheduling methods have been classified in several ways, following different criteria [41, 140, 141, 152]. The main alternatives are load balancing vs. load sharing algorithms, centralized vs. distributed algorithms, and static vs. dynamic algorithms.

Global scheduling algorithms can be classified as being load sharing or load balancing algorithms. The goal of a load sharing algorithm is to maximize the rate at which a distributed system performs work by avoiding that no server node is idle, while load

\(^1\)We use the definition of global scheduling given in [41] and dispatching as synonymous.
balancing algorithms strive to equalize the loads on all the server nodes [89, 141]. If we consider that load sharing objective is to smooth out transient peak overload periods on some of the nodes, a Web switch should aim to share rather than to balance cluster workload. Indeed, absolute stability is not always necessary (and sometime impossible to achieve) in highly dynamic systems such as Web-server systems.

The Web cluster architecture with a single Web switch that receives all incoming requests drives the choice to centralized dispatching policies.

Practically, the main choice for dispatching algorithms deployed by the Web switch lies in the kind of system information the switch uses to make assignment decisions. The policies are grouped into two main classes: static and dynamic; they will be investigated in the next section.

3.3.1 A Taxonomy of Dispatching Algorithms

Dispatching algorithms implementable at the Web switch range from static algorithms that do not consider any system state information to global dynamic algorithms that take into account some system state information while making assignment decisions. A third class of load sharing algorithms, that is adaptive policies, has been widely investigated in literature, where the load sharing policy as well as the policy parameters change on the basis of system and workload conditions [141]. However, to the best of our knowledge, no existing Web cluster uses such an algorithm. Therefore, adaptive policies will not be considered further.

The system state information used in dynamic policies may be either some client information, some server state information, or even a combination them both. As the OSI protocol stack layer at which the Web switch operates the request routing makes available different types of client information to be used in the server selection process, we can broadly classify the dispatching algorithms in content-blind dispatching, if the switch works at TCP/IP level, and content-aware dispatching, if the switch works at application level. Not only this classification allows us to distinguish the client information available at the switch, but also the server information that can be used in the dispatching decision, as it will be explained in the next sections. Therefore, dynamic algorithms can be further classified according to the level of system state information being used by the Web switch:

**Client state aware policies.** The Web switch routes requests on the basis of some client information. Layer-4 Web switches can use only network client informa-
tion 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-based dispatching, the switch can also take into account information on the current contents of each server cache. As we will see in the next sections, the use of server state information at the Web switch may require switch/server communications, depending on the cluster architecture.

**Client and server state aware policies.** The Web switch routes requests on the basis of both client and server state information. Actually, most of the existing client-aware policies belong to this class, because they always use some more or less precise information about the server loads (e.g., server availability).

Figure 3.10 summarizes the taxonomy for dispatching algorithms that we have examined so far. It is to be noted that we assume that static algorithms as well as server state aware policies are deployed only by Web switches that operate at TCP/IP level. Indeed, the use of a more sophisticated architecture (i.e., a layer-7 switch) does not make sense if its benefits are not exploited by the dispatching algorithm.

![Diagram](https://example.com/diagram.png)

**Figure 3.10:** Taxonomy of dispatching algorithms.
The Web switch cannot use highly sophisticated dispatching algorithms because it has to make fast decision for hundreds 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. 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.

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, switch and Web servers of the cluster are the only entities that can collect and exchange load information. Any state information that needs active cooperation from any other Web component cannot be considered, otherwise the previous requirement would be violated.

### 3.3.2 Content-blind Dispatching

In this section, we describe some content-blind dispatching algorithms according to the classification illustrated in Figure 3.10.

**Static Policies**

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 (i.e., with equal probability of going to each 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 of different processing capacities making the assignment probabilistic on the basis of the server
capacity. To this purpose, if $C_i$ indicates the server capacity, the relative server capacity $\xi_i$ ($0 \leq \xi_i \leq 1$) 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 $\varrho$ ($0 \leq \varrho \leq 1$) 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, where 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 [101]. 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 $S_1 S_2 S_3$.

**Client State Aware Policies**

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 elementary client information can be used to provide a simple method of implementing quality of service, by statically partitioning server nodes and assigning groups of clients identified through their IP address to different server partitions.

**Server State Aware Policies**

Two issues need to be addressed first when we consider a server state aware dispatching policy: the selection of a server load index [64] and how to compute the load state information because it may not be immediately available at the Web switch, by reason of the cluster architecture. Indeed, the Web switch can either receive load information from an agent process running on the servers (which periodically evaluates the server load state and transmits it to the Web switch), or monitor their behavior using the traffic that flows through itself. The latter approach is feasible in a two-way cluster
architecture only, as in that instance the Web switch has a full control on the data flow.

The three main factors that affect the latency time are loads on CPU, disk and network resources of the Web server nodes. Typical server load indexes include the CPU utilization, the amount of available memory, the disk or I/O storage utilization, the instantaneous number of active connections, and the object latency time, that is the time each object request spent in the server resources in order to be completed.

Load metrics can be classified, according to the way they are computed, into three classes: input metrics, server metrics, and forward metrics. Input metrics are computed locally at the Web switch and do not require any server cooperation, e.g., the number of active connections. They provide an estimate of the state of the Web servers as seen by the Web switch. Server metrics are computed by each server and transmitted to the Web switch, e.g., CPU utilization, latency time. Forward metrics are information got directly by the Web switch but require that the Web switch emulates requests to the Web servers. For instance, the Web switch can send an HTTP request to each Web server and evaluate the time spent to retrieve the corresponding Web object. Both server and forward metrics typically take longer than input metrics to acquire. Moreover, forward metrics generate additional traffic on the intra-cluster network and can impact on Web server loads; therefore, we will not consider them further.

The selection of the server load metric can depend on the cluster architecture. In one-way cluster architectures, the object latency time cannot be easily tracked at the Web switch, as responses do not flow through the switch on their way back; its evaluation requires a tracking on server and a server-to-switch communication. On the other hand, a two-way switch can easily monitor the latency time of each request, for example by calculating the time elapsed between routing the first byte of the request to the server and sending the first byte of the response out to the client. The same applies for the number of active connections: in a two-way architecture, the Web switch can count this index by analyzing the binding table. On the other hand, in a one-way architecture the information contained in the binding table cannot be as accurate as when responses pass back through the Web switch.

A more fine-grained load index than the number of active connections is the number of processed packets. This can be a useful load index when the number of packets transferred varies greatly from connection to connection, as when either a large file is transmitted or HTTP/1.1 persistent connections are employed in the client/server
interaction.

Other load indexes, such as CPU utilization, need server-to-switch communications (i.e., can be only classified as server metrics). However, CPU utilization might not accurately reflect the real load imposed on the server, especially when large files or transactional objects that require interaction with application or database servers have been requested.

Once a server load index is selected, the Web switch can apply different load sharing schemes. A common scheme is to have the new connection assigned to the server with the lowest load index. For example, in the Least Connections policy, which is usually adopted in commercial products, the Web switch assigns the new connection 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 server heterogeneity [101]. The underlying idea is that servers with greater capacity should support a larger number of active connections: this can be achieved by dividing the number of active connections by the server weight. As a further example, in the Fastest Response policy, the Web switch assigns the new connection to the server which is responding faster (i.e., with the smallest latency time).

The Weighted Round-Robin (WRR) algorithm comes as a variation of the Round-Robin policy. WRR associates each server with a dynamically evaluated weight that is proportional to the server load state. Periodically, the Web switch gathers this information from the servers and computes the weights, that are dynamically incremented for each new connection assignment. Additional information on WRR can be found in [79].

Client and Server State Aware Policies

Client information available at a layer-4 Web switch is usually combined with server state information to provide the so-called client affinity based on the source IP address and the service port number within the TCP header [79, 101]. 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 in the life span of the SSL key, so to avoid time- and resource-consuming for SSL key negotiation and generation. Conversely, an example of a
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.

Figure 3.11 summarizes the classification of the content-blind dispatching policies and shows some policies we have described in this section.

![Content-blind dispatching](image)

**Figure 3.11: Content-blind dispatching algorithms.**

### 3.3.3 Content-aware Dispatching

Layer-7 Web switches deploy content-aware distribution, as they can examine the HTTP request and make the assignment decision on the basis of detailed information about the client.

**Client State Aware Policies**

Since the client information used in the scheduling decision ranges from the URL content, SSL identifiers, to cookies, we can distinguish client state aware policies accordingly. The requested URL can be used either to improve the hit rate in the server caches, to partition the content among servers, or to share the load imposed by different Web services.

In **static cache affinity** policies, the file space is typically partitioned among the server nodes. A hash function can be used to perform the partitioning. This scheme exploits at best the locality of references in the server nodes. However, it ignores load
Section 3.3. Dispatching Algorithms for Web-server Clusters

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, 51]), the server nodes serving those critical files will be more loaded than others.

The requested URL can be used to partition the servers according to the content type they handle with the goal to employ specialized servers for certain type of requests (e.g., dynamic content, multimedia files, streaming video) [157].

A more sophisticated approach to share the load considers content partitioning among the servers according to the file size distribution. The Size Interval Task Assignment with Equal load (SITA-E) policy defines the size range associated with each server in such a way that the total load directed to each server is the same [73]. 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 processing only.

Most problems occur in load sharing 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 server partitioning only. To improve load sharing in Web clusters that provide multiple services, the Client-Aware Policy (CAP) takes into consideration the requested service [39, 40]. Web requests are classified into four main categories on the basis of their impact on main Web server resources: static, lightly dynamic Web publishing services, disk bound services, CPU bound services, and 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. 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 scheduling choice is determined statically once the requested URL has been classified.

The last type of client information on which the Web switch can base the server selection is the so-called “session identifier”, like cookies and SSL identifiers. Based on it, the Web switch assigns all requests from the same client to the same server. Session identifiers provide a powerful means to maintain client affinity at the individual client
granularity, as they avoid the limitations of the source IP address visible in content-blind dispatching. Indeed, the fact that Web proxies are interposed on the client-server path squeezes a large number of users into a very small number of IP addresses. Therefore, when the Web switch uses only the source IP address, clients who are not part of the session are routed to the same server because they are coming through the same proxy.

**Client and Server State Aware Policies**

Dispatching algorithms deployed at layer 7 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, but some client aware policies (e.g., CAP) can be easily supplemented with some server state information.

The **Locality-Aware Request Distribution** (LARD) policy is a content-based request distribution that considers both locality and load balancing [16, 125]. The basic principle of LARD is that all requests for a Web object are directed to the same server node. By so doing, the requested object is more likely to be found into the disk cache of the server node. LARD assigns all requests for a target file to the same node until it reaches a certain utilization threshold. Beyond this point, the request is assigned to a low loaded node, if it exists, or to the least loaded one. A scheme similar to the LARD policy has been implemented also in the HACC cluster [162].

While in LARD policy the Web switch maintains the mapping from a given file to a set of nodes that serve that file, the dispatching policy can also rely on a cache manager with knowledge of all current cache contents. 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 [34]. We refer to this policy as **dynamic cache affinity**.

Figure 3.12 summarizes the classification of the content-aware dispatching policies and shows at the bottom level the policies we have described in this section.

**3.3.4 Dispatching Algorithms Comparison**

In this section, we first compare the algorithms we have classified within content-blind and content-aware dispatching. Then, we compare the two whole categories.
Content-blind Dispatching

Static algorithms are the fastest dispatching solution because they do not rely on any system state information in making the decision. Furthermore, they are very easy to implement. However, these stateless algorithms may make poor assignment decisions due to highly variable service times and resource consumption that characterize Web requests.

Dynamic algorithms have the potential to outperform static algorithms by using some state information in the process of dispatching decision. To that purpose, they require mechanisms that collect and analyze state information, thereby incurring in potentially expensive overheads. Furthermore, the parameters setting of dynamic policies can be difficult in a highly variable Web environment.

For a layer-4 Web switch, server state aware algorithms seem to be the best choice. Specifically, the WRR policy is adopted in real Web clusters because many experiments and simulation results have demonstrated it to compromise simplicity with efficacy at best [39, 79].

Irrespective of the server load index, the least loaded approach tends 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, 110], yet the least loaded approach is commonly used, especially in commercial products.

All server state aware policies face the problem of updating the load information. The intervals between updates of the load information 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].

**Content-aware Dispatching**

Client state aware policies must limit the parsing of client information not to cause excessive overhead on the Web switch. For example, a cookie can be 4096 characters long and this information can be carried on many TCP segments. If the Web switch has to inspect every cookie before assigning the corresponding request, not only the latency time increases but also the Web switch can easily become the system bottleneck.

Content-aware dispatching policies based only on client information provide better performance when they take into account the impact of the service being requested, as in the CAP policy [40]. Client info 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 scheduling decisions. However, they have to take into account at least a binary server information in order to avoid routing the requests to temporarily unavailable or overloaded servers.

The LARD is a performing request distribution policy well designed for Web clusters that provide static content, as it exploits both locality and load balancing. However, its efficacy is reduced when the Web cluster provides both static, dynamic, and secure content [39].

**Content-blind vs. Content-aware Dispatching**

Content-aware dispatching policies can potentially outperform the content-blind ones as they rely on more detailed client information in making the assignment decision. For example, the LARD algorithm shows substantial performance advantages over the WRR strategy when considering static content [16].

On the other hand, content-aware dispatching cannot use too sophisticated information not to severely limit the cluster scalability which already suffers from a more expensive routing.

The complexity of services and applications provided by Web sites is ever in-
creasing as evidenced by the integration of traditional Web publishing sites with e-commerce and transactional sites which combine dynamic and secure services. Therefore, content-aware dispatching should address the service heterogeneity by investigating the combination of client and server information.

3.4 Classification and Summary of Products and Prototypes

In this section, we classify research prototypes and commercial products that implement Web-server clusters according to the architecture taxonomy outlined in the Section 3.2 and illustrated in Figure 3.9. In the classification, we focus on the cluster architecture; as regards the dispatching algorithms, it is to be noted that commercial products usually deploy simple policies, while research prototypes have investigated more performing solutions. We will describe some products and prototypes in details.

In the last four years, the size of Web-server systems market has rapidly expanded: in 1996 it was close to $0 worth, while in 2000 it reached $580M. Therefore, there is a large number of companies that investigate solutions in this field; in this section, we consider the most notable ones. It is to be considered that company names can also change because of merges or acquisitions: small companies have been taken over by big ones that wanted either to enter into this market or to strengthen their position. For example, in June 2000 Cisco Systems [44] acquired ArrowPoint, one of the first companies to commercialize layer-7 Web switches, while in January 2001 NortelNetworks [122] entered this market by acquiring one of the market leaders, Alteon WebSystems.

3.4.1 Layer-4 Solutions

Table 3.2 classifies some commercial products and research prototypes that work at TCP/IP level. Some products, e.g., Linux Virtual Server [101], appear in the table more than once as they can be configured to support more than one cluster architecture.

Hereafter, some of research prototypes and commercial products classified above are described.

LocalDirector. The LocalDirector product from Cisco Systems [44] is an early commercial implementation of layer-4 Web switch based on the NAT approach. It
<table>
<thead>
<tr>
<th>Two-way</th>
<th>One-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet double-rewriting</td>
<td>Packet single-rewriting</td>
</tr>
</tbody>
</table>

Table 3.2: Layer-4 Web cluster proposals.

offers some load sharing policies, among which the least connections, that selects the server with the least number of active connections, and the fastest response, that dispatches the request to the server that was fastest in responding to previous connection requests.

Through the use of a sticky flag, it can support some stateful services, such as SSL. This support is accomplished by directing multiple connections from the same client to the same server within a given period of time (5 minutes by default).

**Magicrouter.** Magicrouter, developed at the University of California at Berkeley, provides an early implementation of a layer-4 Web switch with packet double-rewriting [7]. It uses a mechanism of fast packet interposing, where a user level process, acting as a switchboard, intercepts client-to-server and server-to-client packets and modifies them by changing the addresses and checksum fields. To share the load among the Web-server nodes, three algorithms are considered: round-robin, random and incremental load; the last 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.

**Network Dispatcher.** IBM Network Dispatcher [79, 81] is an extension of the basic
TCP router [55]. Network Dispatcher provides a packet forwarding mechanism, where all the server nodes share the same VIP address. The switch forwards packets, destined for the cluster, to a selected server using its MAC address on the LAN, without modifying the TCP/IP headers. The dispatching policy, used by the switch, can be dynamically based on server load and availability. Specifically, the Network Dispatcher uses a weighted round-robin algorithm to distribute connections among the server nodes. Through a mechanism that IBM calls client affinity (similar to Cisco’s sticky flag), the Network Dispatcher is able to support stateful services.

**ONE-IP.** One of the first implementations of a layer-4 Web switch, based on packet forwarding, was ONE-IP, developed at the Bell Labs [55]. It uses the `if config alias` option to configure the aliasing interface of all Web-server nodes with the VIP address (here called ONE-IP). Two different dispatching methods are supported. With routing-based dispatching, the switch selects the destination server based on a hash function, that maps the client IP address into the primary IP address of a server, and then reroutes the packet to the selected server. With broadcast-based dispatching, the switch broadcasts packets having ONE-IP as destination address to every server in the cluster. Through local filtering each server evaluates whether it is the actual destination by applying a hash function to the client IP address and comparing the result with its own assigned identifier. The main advantage of the ONE-IP approach is that it does not need to keep track of any system state information. On the other hand, the use of a hash function to select the server, based on the client IP address, is its weak point. Although the hash function could be dynamically modified to provide fault tolerance, this approach is not able to adapt to conditions when clients unevenly load the servers. Furthermore, the proposed hash function does not take into account server heterogeneity.

### 3.4.2 Layer-7 Solutions

In Table 3.3 we classify some commercial products and research prototypes that work at application level. Some products herein listed, e.g., Foundry Networks’ Server-Iron [101], have already been considered in Table 3.2 as they can support both content-blind and content-aware routing.

As representative of the various research prototypes and commercial products, we
describe the following two.

**LARD.** The Locality-Aware Request Distribution (LARD) architecture has been developed at Rice University [16, 125]. The request routing occurs at application level and allows the server nodes to send the response packets directly to clients. The incoming HTTP request is accepted by the Web switch, which classifies the requested content and dispatches the request to an appropriate server, selected on the basis of the LARD policy described in Section 3.3.3. The routing is done with the aid of a modified kernel that supports a connection handoff protocol: after the server selection, the Web switch informs the target server of the status of the network connection and the server takes over the connection, communicating directly with the client. Incoming traffic on already established connections is forwarded to the target server through an efficient forwarding module layered at the bottom of the Web switch’s protocol stack. LARD requires that the server operating system is modified in order to support the TCP handoff protocol. On the other hand, the handoff protocol is transparent to the Web server application running on the server nodes, thus allowing the use of any off-the-shelf Web server (e.g., Apache [9]).

To support HTTP/1.1 persistent connections and assign requests in the same connection to different servers, the LARD prototype has been extended with a
back-end forwarding mechanism, that allows the original target node to forward a request to a second server selected by the switch [16].

Central Dispatch. Resonate’s Central Dispatch [135] is, to the best of our knowledge, the only commercial product that enables a direct way from server to client (i.e., a one-way architecture) in a layer-7 architecture. It offers a proprietary solution called Connection Hop, which resembles the TCP handoff mechanism proposed in [16, 125]. Central Dispatch is a distributed software solution that is loaded on the Web switch and on each Web server node. The connection hop operates at the network layer between the network interface card and the TCP/IP stack, thus minimizing the latency on incoming packets. Dispatching of requests that arrive at the Web switch is based on client information and server availability and performance. Upon receipt of the HTTP request, the switch parses the URL to determine content being requested. If more than one server is available to serve the request, the switch selects the least loaded one and transfers the TCP connection to the target server.

3.5 Architectures Combining Basic Solutions

This section presents some possible extensions to the basic cluster architecture described in Section 3.1. Their goal is to improve the system performance through the combination of some cluster components.

3.5.1 Integrated Solutions

In this section, we describe Web cluster architectures that integrate some solutions examined in the previous section. Specifically, we discuss the combination of layer-4 and layer-7 Web switches, in order to overcome the scalability problem that occurs in content-aware routing. We also consider architectures that decouple the two main functions of the Web switch (i.e., packet routing and server selection) among distinct nodes in order to achieve a higher throughput.

Layer-4 Combined with Layer-7 Solutions

Since the additional overhead imposed by content-aware routing can reduce the system scalability by one order of magnitude, solutions that address this problem have been proposed in [17, 143].
Aron et al. [17] designed a prototype in which a front-end layer-4 switch receives client incoming requests and distributes them among the server nodes, which may forward the incoming request to another server node based on the requested content. A distributor component resides on each server node, in such a way that the Web server and the distributor are co-located on the same node. Upon receiving a new TCP connection request, the layer-4 Web switch selects one distributor on the basis of the servers' load. The distributor accepts the connection, parses the client request, and contacts a dispatcher located on the internal LAN for the assignment. If the dispatcher selects a different server from the one the distributor is hosted, the distributor hands off the connection using the TCP handoff protocol to the server chosen by the dispatcher. Then, the server responds directly to the client without going through the layer-4 switch. In case of a TCP connection handoff, the distributor sends a message to the layer-4 switch instructing it to route packets not to the original server but to some other node chosen by the dispatcher.

It is to be noted that in this architecture the second-level dispatching decision is centralized and executed by a single dispatcher, while the second-level routing is distributed and carried out by each distributor component on the server nodes. The motivation is that the processing overhead on a layer-7 switch is caused by the routing and not by the server selection task. However, even if in [17] the authors show that the dispatcher node is not the system bottleneck, a distributed dispatching algorithm carried out by the server node could avoid the communication overhead caused by a centralized dispatcher.

A similar architecture, that examines a combination of content-blind and content-aware routing, has been proposed by Song et al. [143], though with a different goal. Indeed, they describe the prototype of a scalable Web server accelerator, that improves Web server performance by caching data in the main memory. The basic architecture consists of a layer-4 switch in front of a set of Web caches accelerator nodes (instead of Web servers nodes as in the architectures considered so far). In their turn, the cache nodes are located in front of Web server nodes. The layer-4 switch receives incoming requests and routes them to the cache nodes regardless of the requested object. 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 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 responds directly to the client (without passing through the first node or the layer-4 switch).

**Multiple Switches**

The Multi-Node Load Balancing (MNLB) architecture from Cisco Systems [44] separates the packet-by-packet processing functions (changing addresses and port numbers, recalculating checksums) from the server selection action, in order to eliminate the scalability limitations that occur in two-way layer-4 architectures (e.g., Cisco’s LocalDirector). The front-end switches, called *Forwarding Agents*, receive incoming packets from the clients. Multiple switches can be used to spread the load for reliability and scalability reasons (the switch selection can be carried out by the authoritative DNS server for the Web site, as explained in Section 4.2, or through the use of a multicast MAC address.) If the packet belongs to a new connection, the agent forwards it to the *Service Manager* node which makes the dispatching decision based on server information. The manager informs the agent about its decision, so subsequent packets belonging to the same connection are forwarded directly to the target server by the agent without passing through the manager.

### 3.5.2 Multi-tier Architectures

Architectures for e-commerce Web sites are usually structured in layers rather than in a flat architecture. So far, we have consider Web cluster architectures which have one tier only; all the server nodes are connected to the Web switch and are capable of handling requests of any content.

Instead, in a multi-tier architecture there is a Web switch located between the outside world and a set of front-end nodes on which Web servers run. These front-end nodes are capable of handling requests for static content, while they forward requests for dynamic content to back-end nodes, on which application servers run. The Web server nodes and the application server nodes are typically connected to the same network. One dispatching node, referred to as *second-level Web switch*, is in charge of selecting an appropriate application server node for requests that need dynamic processing.

When a Web server node receives a request for dynamic content, it queries the second-level switch to determine which application server should be used to process the request; then, it sends the request to the selected node. The two-tier architecture
can be further extended with a third level of server nodes, composed of database servers which respond to queries originated by the application servers thus providing the content necessary to dynamically generate the requested object. Figure 3.13 shows the main system components comprising a multi-tier cluster architecture. Furthermore, a set of reverse server caches (also called Web server accelerators) is typically used as a front-end to the Web cluster. The caches store frequently accessed Web objects and responds to requests for these objects, relieving the Web servers of that workload [42, 83].

A two-tier architecture has been studied in [164]; here the second-level switch is integrated in each Web server node, called master node, and selects the appropriate slave node, which processes the dynamic request and sends the result back to the master. The slave node selection is based on a prediction model that estimates at the master node the expected cost for processing the dynamic request on each slave node. Similar multi-tier architectures have been also described in [32, 67].

Figure 3.13: Multi-tier cluster architecture.
3.6 Concluding Remarks

Web 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. In this chapter, we have analyzed routing mechanisms and dispatching algorithms suitable for locally distributed Web systems. We have proposed an original taxonomy of Web cluster architectures as well as of dispatching algorithms; we have analyzed the efficiency and the limitations of the different techniques and evaluated the tradeoff among alternatives. To conclude, a touch on directions of future research.

While layer-4 Web cluster architectures may be considered a solved problem, the area of content-aware architectures needs further research. Dispatching algorithms that effectively combine client and server information are desirable. In particular, the differentiation of Web services needs further research.

Furthermore, the scalability problem posed by content-aware routing has not been completely solved yet and distributed dispatching algorithms executed by the server nodes can be a theme of in-depth investigation. Dispatching in multi-tier architectures is an interesting issue that requires additional work to avoid information inconsistency among the multiple switches.
Chapter 4

Geographically Distributed Web-server Systems

Locally distributed Web systems are concerned with improving scalability, performance, and reliability of Web servers. However, hosting a Web site at one single location poses several problems. The first is that client requests and server responses tend to congest the network backbones and clients have no capability to avoid congested network links. The second problem is that upgrading content site infrastructure from a single node to a locally distributed system provides only a limited relief, as the network link of the Web site to Internet may become the system bottleneck. Indeed, apart from the Internet backbone, few Web sites have a wide-area connectivity at or above a fiber optic OC-3 link (155 Mbps). Therefore, the performance bottleneck that limits the scalability of local architectures is not the capability of the servers to generate responses but rather the capability of the network to get data from the server to the client. The third main problem is related to network reliability. If the network zone of the Web site is unreachable or congested, the user cannot access information regardless of the reliability of the Web site. A relatively straightforward method to potentially reduce network impact on users’ response time and to scale to large traffic volumes is distributing Web servers across multiple geographically dispersed locations, namely global scale-out as defined in Section 2.6.

Geographically distributed Web systems, in which a single site name refers to content located at geographically wide-spread physical destinations, are the most scalable architectures to handle the thousand of hits per minute that stress highly accessed Web sites. Indeed, with respect to clusters of nodes that reside at a single location, they can provide better response time by routing client requests to the lowest latency
server for that client; this allows to minimize WAN delays and decrease bandwidth consumption on network links. Moreover, they provide higher scalability, because the cluster wide-area connectivity may become the bottleneck. Geographically distributed Web systems can also provide wide-area failover, that is an increased availability in the face of catastrophic network failures or overloaded Web clusters. However, their more complex architecture poses some interesting challenges which are not to be found in local architectures. A first key issue concerns the routing mechanisms and dispatching algorithms that can be used for the server selection. A second issue, central to the performance of a global architecture, relates to the selection metric and the acquisition of information being used in the selection decision. For example, as server selection aims to minimize user's perceived response time by routing client requests to quickly responding servers, it is fundamental to measure the nearness between clients and servers. A third issue is how to decide about the placement of Web server nodes in strategic Internet locations; this decision can be taken either statically or dynamically in order to make the Web system more responsive to changes in the demand pattern. In this thesis, we do not examine the node placement issue, but proposals that attempt to solve this problem, which is common to proxy caching architectures, can be found in [94, 132].

In this chapter we describe and discuss the various approaches for routing and dispatching client requests in geographically distributed Web systems. We also examine how global architectures and related management algorithms satisfy the scalability and performance requirements of Web services.

The rest of this chapter is organized as follows. Section 4.1 discusses the architectures of geographically distributed systems and presents an original taxonomy of such architectures. Section 4.2 covers the routing mechanisms that can be deployed to route client requests to wide-area dispersed server nodes. Section 4.3 describes dispatching algorithms that aim to achieve load sharing and/or to reduce user-perceived response time. We present a taxonomy of the proposed policies focusing on the dispatching entity and the system information used by each policy. In Section 4.4, we classify various academic and commercial products, according to the architecture of the geographically distributed Web-server system and the routing mechanism used to distribute client requests among the server nodes. Section 4.5 deals with content distribution networks, which represent an alternative solution to deploy global distribution of Web content. Section 4.6 concludes the chapter with some remarks on open research issues.
4.1 System Architectures

A geographically distributed Web-server system is composed by a loosely coupled architecture where the server nodes are scattered over the Internet. Since the object of this thesis are only Web-server systems that fulfil the architecture transparency requirement by providing a single virtual interface to the outside world, Web site mirroring will not be discussed.

Geographically distributed architectures can be differentiated on the basis of the server entity located in each single location. Given multiple server nodes that host a Web site, there are two main approaches:

- a single Web server in each location, that is the Web site is realized on an architecture of geographically distributed Web servers;
- a set of Web servers, typically grouped into a Web-server cluster, in each location, that is the Web site is realized on an architecture of geographically distributed Web clusters.

We call the former solution Distributed Web-server System (DWS), and the latter solution Distributed Web-server Cluster (DWC). DWS solutions have been introduced in Section 3.1, but we have not explored them further in Chapter 3 because the server nodes make typically their IP addresses visible to clients. Indeed, locally distributed Web systems can provide a single virtual interface not only at the site name level but also at the IP address level, which is a valuable characteristic for local architectures. On the other hand, geographically distributed Web systems mask the multiple server nodes only at the site name level (i.e., the multiple nodes are client-visible), because in a geographical context the request routing mechanism requires to identify uniquely each server with its own IP address (we will see in Section 3.2 that there are a few exceptions to this requirement).

In a geographical context, information is typically replicated among the server nodes of the Web system. For this reason, a server node is also referred to as a server replica.

4.1.1 Distributed Web-server System

A distributed Web-server system consists of geographically distributed nodes, each composed of a single server. In this architecture, the request assignment process can
occur in two steps: a *first-level routing* where the client request is transparently directed to a target Web server using different methods that can be either at DNS, network or Web system level as explained in Section 4.2, and a *second-level routing*, which is typically carried out through some re-routing mechanism by each Web server that cannot fulfill a received request. A distributed Web server architecture is illustrated in Figure 4.1, in which how the request routing occurs in the system is intentionally omitted.

![Web server diagram](image)

**Figure 4.1:** A distributed Web-server system.

### 4.1.2 Distributed Web Clusters

In order to achieve a more scalable Web site, a single Web server can be replaced by a Web cluster. Therefore, a distributed Web-server cluster consists of geographically dispersed nodes, each composed of a cluster of servers. A distributed Web cluster has typically a single site name to which correspond multiple IP addresses, one for each Web cluster front-end node. A DWC architecture is depicted in Figure 4.2. For example, a distributed Web cluster solution has been adopted by IBM to handle the official Web site for the 1998 Olympic Winter Games [42, 83].

In distributed Web clusters, there are at least two levels at which the request routing occurs: a *first-level routing* where the client request is routed to a target Web cluster using the same methods available for DWS architectures, and a *second-level routing* deployed by the Web switch component of each Web cluster. In a distributed
Web cluster, the second-level routing corresponds to a routing which is internal to the local Web cluster. We suppose that requests reaching a Web cluster are routed through one of the mechanisms described in Section 3.2. In this chapter, we focus on request routing and dispatching among the Web clusters. A third-level routing can be also carried out through some re-routing mechanism by the Web switch of each cluster or even by each Web server that cannot fulfill a received request.

Figure 4.3 summarizes the taxonomy for geographically distributed Web systems that we have examined so far.

4.2 Routing Mechanisms

In this section, we analyze the request routing mechanisms that apply to geographically distributed systems. We distinguish the mechanisms according to the level at which the decision on client request routing occurs along the path from the client to the Web site (see Section 2.6.2). Therefore, we analyze in Section 4.2.1 DNS-based mechanisms where the routing takes place during the address resolution phase, in Section 4.2.2 network level mechanisms where the network routes the client request, and in Section 4.2.3 routing mechanisms that are deployed at the Web system level by the cluster Web switch or the servers themselves. Since all routing mechanisms can be used by both DWS and DWC architectures, henceforth we refer to a Web
node, which may consist of either a single Web server in DWS architectures or a Web cluster for DWC architectures.

DNS-based as well as network level mechanisms are used only for the first-level routing, while mechanisms deployed at the Web system can be used for every level of routing. The first-level routing is typically centralized at a single entity, while the second-level routing in DWS architectures and the third-level routing in DWC architectures is distributed.

4.2.1 DNS Mechanisms

DNS-based routing is the first solution that has been proposed in 1994 to handle multiple Web servers hosting a Web site and it was originally conceived for locally distributed Web systems [98]. It works intervening on the address lookup phase that occurs at the beginning of the Web transaction. The single site name is mapped to a set of IP addresses corresponding to the Web nodes, thus the authoritative DNS server (ADNS) for the Web site can select one of them for every address resolution request [33]. For each address request reaching the Web system, the ADNS returns a tuple <IP address, TTL>, where the first tuple entry is the IP address of one of the nodes in the Web-server system, and the second entry is the Time To Live (TTL) period during which the name servers along the path from the ADNS to the client cache the mapping. Figure 4.4 shows a distributed Web cluster where the first-level routing is carried out by the ADNS, which translates the site name into the VIP
address of one of the Web clusters.

Although conceived for local architectures, DNS-based routing can scale well geographically and do not present risks of bottleneck. The popularity of this approach for wide-area Web systems is increasing due to the seamless integration with standard DNS and the generality of the name resolution process, which works across any IP-based application (i.e., it satisfies the support to Internet applications requirement).

However, the address caching occurring at various network levels limits the ADNS control on the request distribution among Web nodes as it reduces to a small percentage the address requests that actually need the authoritative DNS server to handle the address request [48, 57]. (Measures on real traces indicate that the requests under direct control of the ADNS responsible for a highly accessed Web site are less than 5% of the total requests reaching the system.) Indeed, along the resolution chain between the client and the ADNS there are several name servers which can hold a valid mapping for the site name. When this mapping is found in one of the name servers on this path and the TTL is not expired, the address request is resolved bypassing the name resolution decision provided by the ADNS. Only address mapping requests made after the expiration of the TTL in all the caches on the path reach the ADNS. The number of address lookups resolved by the ADNS is further reduced because of caching at the client browser level. As a consequence of both network- and client-level address caching, the DNS-based approach can only permit a coarse-grained request routing.
In order to avoid the address caching at network level and allow for more fine-grain request distribution via the ADNS, the latter can specify a very low value for the TTL. This requirement may make the ADNS a potential bottleneck (because it can be overwhelmed by DNS requests), and increase not only the network traffic for address resolutions but also the response time perceived by users [138]. Moreover, low TTL values can be overridden by some non-cooperative intermediate name servers that impose their minimum TTL. It is worth noting that the TTL period does not work on client-level caching.

4.2.2 Network Mechanisms

In this section, we describe some network-level server selection mechanisms that relies on network routers.

Anycasting

In network-layer anycasting, a common IP anycast address is associated to a set of Web nodes that provide the same service [127]. Each anycast-aware router has in its routing table a path to the node that is closest to the router. Thus, the routing protocol applied by anycast-aware routers chooses among the nodes using the routing distance metric, which is usually based on the hop count. As a consequence, the client datagram is routed to the nearest node. This mechanism suffers from the lack of flexibility in the selection criteria because all requests from a given client are routed to the same Web node and it assumes almost static Web node sets. Moreover, it requires infrastructure changes in the Internet. Since we are not aware of any distributed Web-server system based on this approach, we will not consider it further.

A different approach that supports anycasting at application level has been described in [160], in which the authors propose the definition of an anycast domain name that is mapped to an IP address. The application-layer anycasting architecture does not require changes to the network infrastructure; however, it relies upon changes to the client, which must be anycast-aware, and the introduction of anycast resolvers.

OSPF-based Mechanism

In order to deploy this network-based mechanism, the Web nodes must share the same IP address. This configuration needs to be implemented properly in a geographical
network and requires that all Web nodes belong to the same Autonomous System (AS), that is to the same single administrative network authority. This approach relies on the intra-AS routing protocol such as the Open Shortest Path First (OSPF) that is capable of dealing with multiple instances of the same IP address and determines the lowest cost route to a destination IP address from a specific router [97]. When the packet arrives to the router of the destination network, the latter sends the packet to the closest instance of the shared IP address from its point of view.

This approach work well in a stable network environment; however, if the network topology changes during the same TCP/IP session, the routing tables are updated and the client connection with the previously closest server can be broken.

4.2.3 Web System Mechanisms

In this section, we describe some routing mechanisms that can be carried out by the Web server itself or the cluster Web switch to direct a client request to another node. Specifically, we consider a mechanism deployed at TCP/IP level (i.e., triangulation) and three mechanisms that work at application level (i.e., HTTP redirection, URI rewriting, and JavaScript function). It is worth noting that application-level routing mechanisms can be carried out only by layer-7 Web switches and Web servers.

Triangulation

Routing achieved through triangulation means that the client continues to send packets to the first contacted node. This node routes client packets to the target node, which responds directly to the client. The routing takes place at the TCP/IP level and can be based either on packet tunneling, which has been described in Section 3.2.1, or on packet rewriting. In this solution, the first contacted node modifies each packet by rewriting the destination IP address with that of the target node, which in its turn rewrites the response packets with the IP address of the first contacted node. In packet rewriting (also known as geographical TCP routing), to identify that the packet is coming not from a client but from a redirecting node, special shadowed IP addresses at each node are necessary. Therefore, the redirecting node rewrites the packet using the shadow IP address of the target node and not its known IP address. Figure 4.5 illustrates the traffic flow that characterizes triangulation.
HTTP Redirection

The HTTP protocol standard, starting from version 1.0, allows a Web server to respond to a client request with a specific status code (that is, 301 or 302) in the response header that instructs the client to resubmit its object request to another node in order to fulfill it [28, 65]. The built-in HTTP redirection mechanism supports only per URI-based redirection. The status code 301 (Moved Permanently) specifies that the requested resource has been assigned a new permanent URI and any future reference to this resource will use the returned URI, while the status code 302 (corresponding to Moved Temporarily in HTTP/1.0 and to Found in HTTP/1.1 protocol specifications) instructs the client that the requested resource resides temporarily under a different URI. Figure 4.6 shows the flow of requests and response with HTTP redirection.

An advantage of HTTP-level redirection is that replication can be managed at a fine granularity level, down to individual Web pages. Furthermore, it allows content-aware redirection, as the redirecting server receives the HTTP request and therefore it can take into account the content of the request in selecting an appropriate node. The principal disadvantage is that this mechanism introduces an extra round-trip time into the request processing, as every HTTP redirection requires the client to initiate a new TCP connection with the destination node. For geographically distributed Web
systems, this extra round-trip time increases the network component of the response time; however, this augment could be compensated by a reduction in the response time component due to the server. A minor drawback is that the user can note down in her bookmarks the address of the new server to which it has been redirected, thus defeating the mechanism. Indeed, Web browsers do not handle redirection as specified by HTTP standard. According to it, the browser should display the originally requested URL as well as bookmark it after a temporary redirection message; instead, most browsers display and bookmark the redirected URL.

URL Rewriting

Under URL rewriting, the redirecting node is able to change dynamically the links for embedded objects within the Web page that it serves to point to another node. The drawback of URL rewriting is that it introduces additional load on the redirecting Web node as each Web page has to be dynamically generated in order to contain the modified object references.
JavaScript Function

The redirecting node returns to the client the complete response that contains also a JavaScript function in the header of the returned HTML page. The JavaScript function, executed by the client, replaces the URL of the redirecting node with that of the destination node for subsequent requests [145]. This approach eliminates the double round-trip time of HTTP redirection. However, it requires that the client is JavaScript-enabled, and that all `<a href>` tags in the HTML page contain the Web node name.

4.2.4 Routing Mechanisms Comparison

Figure 4.7 summarizes the taxonomy for global routing mechanisms that we have discussed in this section; it details further the taxonomy previously depicted in Figure 4.3. In the rest of this section, we compare the different approaches identified for routing in geographically distributed Web systems.

![Detailed taxonomy of geographically distributed architectures.](image-url)
Section 4.2. Routing Mechanisms

DNS-based routing determines the destination of client requests through address mapping. The popularity of DNS-based routing for globally distributed architectures is increasing due to the transparency and generality of the name resolution process (e.g., F5 Networks’ 3-DNS [63], Nortel Networks’ Web OS GSLB [122], Resonate’s Global Dispatch [135]). Indeed, most commercial products deploy the first level routing by enhancing the ADNS capabilities. However, this approach permits only a coarse-grained request distribution among the Web nodes because it acts on address request. Moreover, caching at name servers and client browsers avoids contacting the ADNS. Although the setting of TTL to low values allows for more fine-grained request distribution, it may limit general applicability (because of the presence of non-cooperative name servers), make the ADNS a potential bottleneck, and increase considerably the Web access latency perceived by users.

To avoid a failed ADNS disabling the system, multiple name server resource records (which are pointers to other name servers) can be returned to the client local name server, so it can still resolve the address mapping by querying another ADNS whether the first contacted ADNS does not respond.

Multiple name server resource records can be also used to route the client to the best Web node. This technique, called the DNS round-trip times method [6], is based on the current DNS standard and takes advantage of the fact that each local name server, when querying a remote name server, measures the round-trip time of packets to that server. Over time, the local name server queries all the name servers and measures the round-trip time of their responses. As a consequence, the local name server is able to detect the fastest name server in its list. Unlike classic DNS-based Web systems in which the ADNS serves all IP addresses of the Web nodes, the DNS round-trip times technique requires that:

- each ADNS serves an IP address only;

- each ADNS is located close to the Web node of which it provides the mapping.

Although the DNS round-trip times method does not entail any modification of the network infrastructure, it is necessary for it to work efficiently that the local name servers are able to support it. Therefore, we will not explore further this technique because it relies on Internet components which are not under the control of the Web site technical management.

To avoid DNS problems, the routing can be performed by another centralized entity that receives all incoming requests and distributes them among the Web nodes
through HTTP redirection (e.g., Cisco DistributedDirector [44]). However, this solution duplicates the number of necessary TCP connections and can cause the dispatching entity to become the system bottleneck. A better solution to address DNS issues is to add a second-level routing carried out by the Web nodes through a Web system-based routing mechanism. Indeed, we do not consider these mechanisms to be fine for first-level routing, however they can be a useful method to augment DNS-based routing capability. Although a first-level distributed routing avoids introducing in the system a single point of failure, it may augment the response time perceived by the user and require the exchange of a considerable amount of system information among the dispatching entities.

The main disadvantage of triangulation is the overhead imposed on the redirecting node, as it continues to forward data packets to the destination node. That is to say, the triangulation mechanism does not allow the redirecting node to completely get rid of the redirected requests. Network traffic along a congested client-redirecting node path may also increase the response time. Moreover, as triangulation is a content-blind routing mechanism, it does not allow fine-grained dispatching when the Web transaction is carried out on a persistent connection and it requires full server replication.

Unlike triangulation-based solutions, the other Web system mechanisms (i.e., HTTP redirection, URL rewriting, and JavaScript function) do not require the modification of packets reaching or leaving the Web-server system as they are implemented at the application level and can take into account the requested content in the dispatching decision thus allowing a fine-grained re-routing. However, their use limits the service to HTTP requests only. It is worth noting that application-level Web system mechanisms must avoid ping-pong effects that occur when an already re-routed request is further selected for reassignment. This can be achieved by setting a cookie when the request is first redirected and inspecting the cookie prior to decide about reassignment. The triangulation mechanism is free from these side effects as the destination node can deduce if the request has been already re-routed by simply inspecting the source packet address.

The HTTP redirection is fully compatible to any client software. However, its use may determine an increase in response time and network traffic, since a redirected Web page request requires two TCP connections prior to be serviced.

Table 4.1 outlines and summarizes the features and tradeoffs of the various routing mechanisms we have discussed in this section.
### Table 4.1: A summary of global routing mechanisms.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Data flow</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS</td>
<td>Direct</td>
<td>Simple, general applicability</td>
<td>Coarse-grained control, content-blind, limited scalability</td>
</tr>
<tr>
<td>OSPF</td>
<td>Direct</td>
<td>Network support, general</td>
<td>Network instability, content-blind</td>
</tr>
<tr>
<td>Triangulation</td>
<td>Triangular</td>
<td>Simple</td>
<td>Redirecting node overhead, TCP-grained control, content-blind</td>
</tr>
<tr>
<td>HTTP redirection</td>
<td>Redirection</td>
<td>Simple, fine-grained control, content-aware</td>
<td>Transmission overhead, HTTP only</td>
</tr>
<tr>
<td>URL rewriting</td>
<td>Redirection</td>
<td>Fine-grained control, content-aware</td>
<td>Server overhead HTTP only</td>
</tr>
<tr>
<td>JavaScript funct.</td>
<td>Redirection</td>
<td>Fine-grained control, content-aware</td>
<td>Client support HTTP only</td>
</tr>
</tbody>
</table>

### 4.3 Dispatching Algorithms

In this section, we describe policies applicable for distributing requests in global Web-server systems. Dispatching algorithms for a geographical context have to address new challenging issues with respect to algorithms working in a local environment. Indeed, load fluctuations which are typical of Web workload are amplified by the geographical context [85]. Requests arrive in bursts, clients connected to the Web site are not uniformly distributed among the Internet domains, network conditions vary with the time of the day and the day of the week, world time zones are another cause of heterogeneous source arrivals.

#### 4.3.1 A Taxonomy of Dispatching Algorithms

As the dispatching algorithms for locally distributed Web systems, policies for distributing requests in a geographical system range from static algorithms that do not consider any system state information to dynamic algorithms that take into account some system state information while making assignment decisions. On the other hand, while the presence of a Web switch drives the choice for centralized dispatching policies in a local environment, both centralized and distributed algorithms can be applied to a geographical context. Indeed, the first-level assignment that is typically carried out by the authoritative DNS server relies on centralized algorithms, while the second (or third) level assignment deployed by some Web system component is typically distributed. As a consequence, we broadly classify dispatching algorithms
according to the level at which they are deployed. Therefore, in Section 4.3.3, we describe centralized DNS-based dispatching algorithms and in Section 4.3.4 we discuss some distributed dispatching algorithms carried out by the Web nodes.

A further level of classification stems from the system state information used by dynamic dispatching policies, which may be either some client information, some server state information, or even a combination of both information. Figure 4.8 summarizes the taxonomy for dispatching algorithms that we have examined so far.

![Diagram of dispatching algorithms]

Figure 4.8: Taxonomy of dispatching algorithms.

Since a key feature of both centralized and distributed dispatching algorithms for geographical systems resides in their ability to route clients’ requests to the “closest” server, we first discuss the concept of network proximity in Internet and describe some methods and metrics that can be used by the dispatching entity (i.e., DNS and Web nodes) to evaluate it.

### 4.3.2 Network Proximity

Significant research efforts have addressed the problem of evaluating network proximity in the context of server selection and placement [38, 59, 72, 103, 123]. Indeed, geographic proximity between two nodes in Internet does not necessarily equate with their network proximity.

Network measurements can be divided into active measurements, which inject additional traffic in the network in order to detect current network conditions (e.g., network probe packets) and passive measurements, which are based on the analysis
Section 4.3. Dispatching Algorithms

of traces as well as routing tables.

Network proximity metrics can be classified according to the information being used to evaluate client to server proximity. The main alternative is static metrics vs. dynamic metrics.

The first static metric we consider relies on geographic distance. It assumes that geographic distance approximates network latency. To obtain the geographic location of a node, a whois database server can be queried. However, this approach is time consuming because the access to the whois database may be slow. Although this drawback can be addressed by maintaining information on the server, the basic assumption on which this approach relies has been contradicted by several studies (e.g., [38]). A second type of static information is based upon network topology and configuration such as number of hops and base bandwidth of a connection. The traceroute utility can be used to measure the number of network hops between a client and a server [97]. The number of hops can be considered a stable information more than a truly static information; indeed, it has been shown that about 90% of routes remains stable for at least six hours [128]. The hop count can be also measured in terms of Autonomous System (AS) hops using the Border Gateway Protocol (BGP) routing tables which are maintained by routers to route traffic across the Internet. However, measuring AS hops may be a gross approximation as it does not take into account the inter-AS topology; moreover, it requires to access routing information from routers. As a consequence, the number of network hops is a more meaningful proximity metric than the number of AS hops [123]. A tool to measure the base bandwidth is bprobe [38].

Static metrics are simple to obtain but they are not able to guarantee the selection of the closest server as “all links are not created equal” [123]. Indeed, static measures do not provide information on current network conditions, thus not allowing the system to react to changes in network performance. Dynamic evaluation of network proximity is based on measuring directly the latency between client and server. Round-trip times of packets, available bandwidth, and latency time of HTTP requests are three possible ways. The round-trip time is measured through the ping utility. Unlike the number of hops, it reflects the network load on the path between client and server; therefore, it should be used when trying to reduce user response time. A tool to measure the available bandwidth is cprobe [38].

An open issue related to static and dynamic metrics regards the correlation of network hop count and round-trip time: old measures collected six years ago found
that it was close to zero, while recent measures have found it to be reasonably strong (i.e., close to 50%) [123].

4.3.3 DNS-based Dispatching

When the authoritative DNS server provides the address mapping, it can use various dispatching policies to select the best server, ranging from simple static round-robin to more sophisticated algorithms that take into account both client and server state information.

Static Policies

DNS-based static policies correspond to those discussed in Section 3.3.2 for locally distributed Web systems. The Round-Robin DNS (DNS-RR), early implementation of this approach [98], maps IP addresses in a circular way. The load distribution under DNS-RR policy is unbalanced because the address caching mechanism lets the ADNS control only a very small fraction of requests. Indeed, as a result of address caching, a burst of subsequent client requests from a particular domain are mapped to the same server during the TTL period, so leading to degraded performance. An uneven distribution of client requests from different domains leads imbalance because many clients from a single domain can be assigned to the same Web node [48, 57]. Additional drawbacks result because the algorithm ignores both server capacity and availability. With an overloaded or non-operational server, no mechanism can stop the clients from continuing to try to access the server by its cached address. The DNS-RR policy’s poor performance needs research into alternatives DNS-based dispatching algorithms that require additional system information.

Client State Aware Policies

Two kinds of information can come at the ADNS from the client side: the typical load that arrives at the Web-server system from each connected domain and the client’s network location.

To face the uneven distribution of client requests among the domains and the limited control of the DNS on the requests assignment, DNS dispatching policies based on the estimation of the hidden load weight were proposed in [48]. This definition denotes the average number of data requests sent from each connected domain to the Web-server system during the TTL caching period which follows an address mapping
Section 4.3. Dispatching Algorithms

request. Proposed dispatching algorithms, which chiefly use this information to assign requests to the most appropriate server, try to identify the requesting domain and the hidden load weight imposed by this domain. One example of these algorithms is the **multi-tier Round Robin** policy, which uses different round-robin chains for requests in domains of different hidden load weights. Other dispatching policies based on the hidden load weight, as well as **adaptive TTL** algorithms that adjust the TTL value to each address request on the basis of the request rate of the source domain will be discussed and evaluated in Chapter 5.

The other client information that the ADNS can use to select the appropriate Web node regards the **client location**. To limit the network latency component in the user’s perceived response time, most commercial DNS-based Web systems evaluate client-to-server network proximity, so that the ADNS can return the IP address of the Web node which is estimated to be closest to the client (e.g., Cisco’s DistributedDirector [44], F5 Networks’ 3-DNS, Resonate’s Global Dispatch [135]). The ADNS can determine the most suitable Web node on the basis of either the relative geographical distance, the relative topological proximity (i.e., number of network hops or number of AS hops) or client-to-server link latency. The latter measure requires the cooperation of the Web nodes. The client location is approximately represented through the IP address of the client’s local DNS server. Therefore, the network proximity is evaluated between the location of the client’s local DNS server and the location of each Web node.

Since DNS-based scheduling is centralized, the ADNS has to maintain a central database of network proximity between each Web node and known local name servers. When dynamic proximity metrics are used, each Web node has to determine the network distance from itself to the client’s local name server and communicate it to the ADNS.

The main issue specific to DNS-based selection based on network proximity regards the identification of the client’s network location. When selecting the Web node with the best network proximity to the client, the ADNS assumes that the client location is represented by the IP address of the client’s local name server that has originated the address request, as this is the only location information contained in the DNS query. This assumption is based on the hypothesis that the client and its local name server are located near each other, in terms of network proximity. However, this assumption does not always lead to an accurate estimate, because the client and its local name server are often quite distant from each other, as quantified in [138]. Moreover, the ADNS sees the client’s local name server only if all intermediate name
servers issue iterative name resolutions. As a consequence, DNS-based mechanisms cannot accurately pinpoint the client's network location.

**Server State Aware Policies**

Knowledge of server state conditions is essential for a highly available Web-server system to exclude servers that are either unreachable because of faults or high loads. A common scheme that uses server state information is to assign the new connection to the Web node with the lowest load. However, in [48] it was shown that policies based on detailed server information (e.g., present and past load) cannot avoid overloading some Web node while under-utilizing others. Indeed, the load information does not capture the TTL effect, that is future arrivals due to past address resolutions. Because of that, the server load information becomes obsolete quickly and is poorly correlated with future load conditions.

Moreover, in a geographical context dispatching policies based only on server load may cause a client to be assigned to a Web node which is half around the world because that node is lightly loaded, thus increasing the network component of the response time.

**Client and Server State Aware Policies**

Client information is usually combined with server state information to provide at least server availability and to avoid overloaded Web nodes. As an example, since the hidden load weight may not always be a sufficient means to predict the load conditions at each Web node, a simple asynchronous alarm feedback from critically loaded servers is introduced [48]. This allows the ADNS to exclude them from further assignments during the critical period. A similar alarm mechanism can be also integrated into adaptive TTL algorithms, as explained in Section 5.3.

Figure 4.9 summarizes the classification of the DNS-based dispatching policies and shows at the bottom level the policies we have described in this section.

### 4.3.4 Web System Dispatching

To address DNS centralized dispatching issues and achieve a fine-grained assignment control, it is possible to add a second (or third, depending on the system architecture) level of dispatching; in the following we refer to this dispatching as the redirection algorithm.
Figure 4.9: DNS-based dispatching algorithms.

The redirection algorithm is typically distributed and carried out by Web servers through some re-routing mechanism, which is typically HTTP redirection or triangulation. We identify three main components of a redirection strategy: the activation policy, the selection policy, and the location policy [141].

The activation policy determines whether a Web-server node has to activate the redirection process. The redirection can be triggered either on client-demand or on server-demand. The first means that the redirection is activated on the basis of the client request (e.g., the type of content being requested, the client location), while in the latter instance the server load drives the redirection process. Redirection on server-demand typically depends on a load threshold. When the server load reaches a predetermined threshold, the server starts to redirect client requests and ends when its load returns below the threshold.

Once decided that the server has to activate the redirection process, the selection policy determines which requests have to be redirected. The reassignment is usually non-preemptive that is, only new requests that have not yet been served can be selected for transfer. When the activation policy is triggered on client demand, the selection policy is encapsulated within the activation policy, while it is a distinct policy when the redirection is triggered on server-demand.

Finally, the location policy selects the Web-server node to receive the redirected request. Location policies also can be classified on the basis of the system information used to determine the new target server, ranging from simple stateless ones such
as Random and Round Robin to more sophisticated algorithms ones that take into account both client and server state information, such as server load and/or network proximity.

Only few redirection algorithms have been investigated until now [7, 18, 100, 146]. Most of them are based on a server-based activation policy [18, 100, 146] and either a stateless round-robin [146] or least loaded server location policy [18, 100]. A redirection algorithm having a client-based activation policy and a location policy that uses server load and network conditions has been studied in [7]. None of the proposed redirection algorithms that activate the redirection process on server-demand explore the selection policy component. That is to say, under these algorithms when a server reaches the load threshold, it starts to redirected all incoming requests until the load drops below the threshold. In Chapters 6 and 7 we will propose and evaluate some redirection algorithms complementary to those above described. In particular, we will explore selection policies more sophisticated than the naive approach that redirects all requests in Chapter 7.

4.3.5 Dispatching Algorithms Comparison

A key issue common to all stateful dispatching algorithms regards the choice of the best interval for obtaining estimate about system state information at the dispatching entity. To obtain an accurate estimate of server load conditions as well as of network proximity based on dynamic metrics, measurements have to be conducted frequently because system conditions can change considerably over a short period. On the other hand, as transmission times cannot be neglected in a geographically distributed Web system, a tradeoff between information and network overhead has to be pursued to set the update interval.

The main drawback of dispatching algorithms based on network proximity is that they are not able to react immediately to heavy load fluctuations of Web workload that are amplified by the geographical context. Indeed, a geographically distributed Web site that tends to serve closest requests only may risk to be highly unbalanced because the amount of request from an Internet region is strictly dependent on day time. The consequence of time zones and proximity algorithms alone is to have one or two highly loaded nodes in two locations and other almost idle nodes. We will analyze this issue in Chapter 7 and propose some solutions. Here, we summarize our proposal. We will integrate the DNS-based dispatching with a distributed assignment activated by each Web server. This further dispatching level allows an overloaded server to
easily shift away some portion of load assigned by the first level dispatching. We will demonstrate that this dispatching level is necessary to guarantee scalability and load sharing in geographically distributed Web sites, and to enhance quality of Web services by augmenting the percentage of requests with guaranteed response time.

4.4 Classification and Summary of Products and Prototypes

In this section, we classify research prototypes and commercial products that implement geographically distributed Web-server systems according to the architecture taxonomy outlined in Section 4.2 and illustrated in Figure 4.7. In the classification, we focus on the system architecture; we also describe some products and prototypes in details.

4.4.1 Distributed Web-server Systems

Table 4.2 classifies some commercial products and research prototypes that realize distributed Web-server systems with one or two levels of routing.

<table>
<thead>
<tr>
<th>One-level</th>
<th>Two-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS</td>
<td>Triangulation</td>
</tr>
<tr>
<td>Cisco’s DistributedDirector [44] Ibm named [136]</td>
<td></td>
</tr>
<tr>
<td>12-DSI [27] SunSCALR [142] [35, 48] Chapter 5</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Distributed Web-server system proposals.

Among research prototypes and commercial products classified above, we describe the following.

DistributedDirector. The DistributedDirector product from Cisco Systems [44] uses a centralized single-level dispatching algorithm: each request reaches a dispatcher that directs it to the available Web server closest to the client, where the network proximity is determined through a combination of Border Gateway Protocol (BGP) and Interior Routing Protocol (IGP) routing information (i.e.,
the number of AS hops from BGP routing tables and the number of hops within the AS from IGP routing tables) and round-trip latency. In the selection process, the algorithm considers also the server availability.

This product can be configured as an enhanced authoritative DNS server which is able to evaluate client-to-server proximity thus returning the IP address of an available server close to the client’s local DNS name server, or as a centralized dispatcher that adopts HTTP redirection. In the latter instance, the DistributedDirector accepts the client TCP connection, receives the HTTP request, determines the most appropriate Web server based on the relative client proximity to servers (in this working mode, the client address is not hidden by the local name server one) and redirects the client to that server through the 302 status code of the HTTP protocol. The communication of distance metrics between router agents at each location is carried out through the proprietary Director Response Protocol.

When configured to work as an authoritative DNS server, the DistributedDirector requires the TTL to be set to zero seconds to prevent the address mapping from being cached by other name servers. As pointed out in Section 4.2.1, the reduction of the validity of cached address mappings may shift the performance bottleneck from the Web system to the DNS infrastructure. DistributedDirector can also route requests among distributed Web clusters which use Cisco’s LocalDirector, even if no specific communication between the LocalDirectors at each location and the centralized DistributedDirector is provided. DistributedDirector requires Cisco routers at every server location plus BGP peering and it does not take into account current server load information.

**SWEB.** The Scalable server World Wide Web (SWEB) system [8] realized at the University of Santa Barbara is a two-level routing architecture where the first level is centralized and based on DNS routing, while the second one is distributed and carried out through HTTP redirection. This solution was originally conceived to deploy a Web cluster. Client requests are initially assigned from the ADNS in the round-robin manner. Each server in the system may reassign a received request to any other server though HTTP redirection. To avoid a ping-pong effect, every request can be redirect only once. The decision to serve or to redirect a request is based on the minimization of the response time criteria, which is determined by taking the server computing capability and Internet bandwidth
into account. Such an assignment implies an overhead of communication among the servers, as every server periodically communicates some information about its current load to the others. However, such additional cost can be considered negligible with respect to the network traffic due to client request.

**Apache-DC.** The Distributed Cooperative Apache (DC-Apache) system, developed at Arizona University [100], has been designed and implemented atop of Apache Web server [9]. It is a distributed single-level solution based on dynamic manipulation of the links embedded in a Web page which is carried out by Web servers. The DC-Apache system also supports dynamic document replication to solve the hot spots problem and to maintain data consistency. The dispatching algorithm is based on global server load information which are piggybacked onto existing HTTP transfers, thus avoiding additional traffic between servers. The main disadvantages of this solution are its flat architecture and the lack of any geographical information in the dispatching algorithm.

### 4.4.2 Distributed Web Clusters

Table 4.3 classifies some commercial products and research prototypes that realize geographically distributed Web clusters with either two or three routing levels.

<table>
<thead>
<tr>
<th>DNS</th>
<th>Two-level</th>
<th>Three-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cisco’s DistributedDirector [44]</td>
<td>Radware’s WSD-DS [134]</td>
<td>Radware’s WSD-DS [134]</td>
</tr>
<tr>
<td>F5 Networks’ 3-DNS [63]</td>
<td>Cisco’s CSS 11000 [44]</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>Eddie [61]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Foundry Networks’ GSLB [66]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM Network Dispatcher ISS [81]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nortel Networks’ Web OS GSLB [122]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HydraWEB’s HydraHydra [80]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coyote Point’s Envoy [50]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radware’s WSD-NP [134]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resonate’s Global Dispatch [135]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel’s NetStructure</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-site Traffic Director [82]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Distributed Web cluster proposals.

Most commercial products integrate global dispatching functions with those provided by their corresponding products for local dispatching. The first-level assignment among the Web clusters is typically carried out by extending the basic DNS tech-
nique to include some proximity and/or load information in the dispatching strategy. Among research prototypes and commercial products classified above, we describe the following.

**3-DNS.** The 3-DNS product from F5 Networks [63] is a device that perform global distribution via the DNS mechanism. It uses BIG/ip devices in front of Web clusters as agents to report about cluster health and load. Each BIG/ip can also measure the round-trip latency to clients and the packet drop rate. The latency is measured by using probes from each cluster to the client’s local name server. Dispatching algorithms supported by 3-DNS include the selection of the closest cluster in which the network proximity is measured either as a topological distance or round-trip time.

**Eddie.** Eddie [61] is an open-source project supported by Ericsson. Global distribution is achieved through an enhanced ADNS, while local distribution in the Web cluster is based on an HTTP gateway. The global dispatching algorithm returns the closest and least loaded Web node. The first time a client queries the ADNS, it returns the least loaded server, while for the assignment of subsequent requests coming from the same local name server the ADNS can also use the proximity information, which is based on round-trip times estimated from the Web nodes to the client’s local name server.

**Radware.** Radware offers two global solutions: Web Server Director for Network Proximity (WSD-NP) and Web Server Director for Distributed Sites (WSD-DS) [134] which are integrated with WSD Pro local solution. WSD-DS was designed for Intranet applications where user location is known. WSD-DS is a three-level dispatching architecture, where the first dispatching takes place at the ADNS in a round-robin fashion, the second level dispatching is local at each the Web cluster, and the third level dispatching allows to redirect client requests to other Web clusters through either HTTP redirection or triangulation based on packet rewriting. The activation of the request re-routing is very simple and based on a predetermined load threshold, which considers the number of users that each cluster can serve. Once the Web cluster reaches this threshold, it starts redirecting requests until its load drops below the threshold. The proprietary *Load Report Protocol* allows Web clusters to communicate their status and traffic load to each other.
Section 4.5. Content Distribution Networks

WSD-NP acts as an authoritative DNS server and adds proximity detection, allowing the ADNS to select a Web cluster through a combination of network hops and round-trip latency. To provide proximity information to the WSD-NP, Web clusters use the proprietary Proximity Report Protocol. As most commercial product, these Radware solutions require to set the TTL to zero seconds to prevent address caching.

4.5 Content Distribution Networks

A quite different solution to geographical distribution of Web content is provided by several companies that offer global delivery services such as Akamai [2], Adero [1], Digital Island [58], and Mirror Image [109]. When using this service, Web site administrators become content providers and delegate the responsibility for wide-area content distribution to an outsourcing content-delivery service company that owns a set of geographically dispersed servers (or caches). They constitute the so-called content distribution network (CDN) containing replicas of the objects, which are usually images or streaming objects. Like in geographically distributed Web-server systems, the content of the Web site is available from multiple servers. However, the content provider does not administrate the client distribution among the servers but delegates this task to a company that manages a CDN. Service level agreements are contracted between the high-volume content provider and the service company.

A content distribution network makes copy of demanding content (popular and/or bandwidth intensive such as media streams) on multiple content delivery servers distributed through the Internet. Servers belonging to a CDN are usually located at the network edges (i.e., close to end users), thus reducing the so-called first mile (bandwidth capacity of a Web site to the Internet) bottleneck. Each server performs the function of a conventional cache, storing a copy of the content that the service company delivers. By serving content requests from much closer to the users, content delivery systems aim to reduce the potential for Web server overloads and Internet delays.

The request routing is performed through a mapping service typically provided by the ADNS of the domain containing the requested content. The DNS mechanism can be integrated with a URL rewriting mechanism carried out by the Web site servers which work in cooperation with the company servers to dynamically change URL references [68, 86, 87]. The mapping service aims to redirect client requests
to a nearby company server with low-medium utilization, thus avoiding the need to traverse the Internet. As DNS-based routing cannot distinguish the requested content, its integration with the URL rewriting mechanism provides a more fine-grained and content-based routing.

Content distribution services based on the DNS mechanism may require an additional name lookup during the Web transaction (in order to resolve the name of the server belonging to the content distribution network). This may cause a considerable DNS overhead [138], as name resolution time can be substantially longer than network round-trip times [47]. On the other hand, a geographically distributed Web-server system does not require an additional name lookup (care must be taken to let Web system-based mechanisms use the server IP address instead of their names). Some content distribution networks (e.g., Akamai [2]) force clients to retrieve embedded Web objects from multiple servers (e.g., the HTML file from one server and embedded images from another server), so the client cannot benefit from the use of persistent connections [87]. Moreover, content distribution services perform mainly wide-area replication of static content, while Web-server systems can also deploy dynamic and secure content distribution.

We describe some companies that manage content distribution networks.

**Adero.** Adero’s GlobalWise Content [1] is a DNS-based content distribution service that requires only a change to the ADNS of the Web site to make it point not the IP address of the Web server site but to a set of Adero GeoTraffic Managers (GTM) which act as ADNSes for the Web site name. So, the client’s local name server contacts an Adero GTM during the address resolution phase and no further address resolution are needed to obtain the Web page. The GTM returns to the requesting client’s local name server the IP address of the nearest Adero server in terms of network topology. The Adero GlobalWise Network is a worldwide network of multi-server nodes, where each node is optimized for the particular services that it runs.

Content is spread among the servers belonging to the Adero GlobalWise Network through a multicast push technology.

**Akamai.** Akamai Technologies, founded by some MIT professors in August 1998, is the market leader in content distribution networks. The Akamai Network System consists of a large number of distributed Web servers. In November 2000, Akamai manages more than 6000 servers located in 335 networks and 54
countries [2]. The servers are typically configured in groups of five with two Ethernet switches.

Akamai’s FreeFlow uses primarily the DNS mechanism to re-route clients from the content provider Web site to the FreeFlow server closest to the client. All content provided by Akamai is part of the Akamai name domain. When the Akamai ADNS is contacted for an address resolution, Akamai considers the location of the client’s local name server as well as information about network conditions and Akamai servers load to route the client request to an appropriate Akamai server. To redirect the client request to the Akamai ADNS, Akamai requires the cooperation from the origin Web site which has to dynamically change the original URL references to let them point to the name space controlled by Akamai. This process is called Akamaiization and entails the transformation of a URL into an Akamai Resource Locator (ARL) at the content provider Web site. Content is spread to the Akamai servers by using a demand-based pull mechanism.

4.6 Concluding Remarks

Geographically distribute Web-server systems are the most promising architecture to manage highly accessed Web sites that have to guarantee performance scalability and reliability. In this chapter, we have proposed an original taxonomy of global architectures as well as of routing mechanisms and dispatching algorithms that can be used to manage the Web system.

Our analysis has pointed out that DNS-based architectures can easily scale up to manage geographically distributed Web-server systems, as highlighted by the large number of commercial products for wide-area load balancing and content distribution networks that rely on DNS-based routing. Our analysis has shown, however, that DNS-based dispatching is constrained by the limited number of address resolution that the ADNS can perform to prevent risks of bottleneck in the DNS infrastructure. This issue creates a challenge to DNS-based dispatching algorithms that we will address in the next chapter.

However, DNS-based dispatching is intrinsically limited because the address resolution is typically performed once at the beginning of the Web transaction. To overcome also sudden surges of load which are amplified by the geographical context, we investigate in Chapters 6 and 7 two and three-level Web system architectures in
which DNS dispatching is integrated with redirection carried out by Web servers.

In the following, we discuss some directions for future research that will be not addressed in the following chapters. Network proximity needs further investigation as its evaluation is central to reduce the effect that routing has on user-perceived response time.

Another area that needs further research regards content distribution networks for both routing mechanisms and request dispatching algorithms. Most current CDNs deploy only distribution of static content because dynamic content is not usually cacheable. Differentiation among requested services requires also a content-aware request routing which cannot rely on DNS-based mechanisms only. Furthermore, request dispatching is an interesting issue that requires further work. Indeed, it seems that current CDNs do not choose the best server but they aim only to avoid notably bad servers [86].
Chapter 5

Load Sharing through DNS-based Mechanisms

In this chapter, we focus on scalable Web-server systems where the client request assignment decision occurs at the Domain Name System (DNS) level during the address lookup phase, that is when the Web site name is translated into the IP address of one of the nodes of the Web-server system. Address mapping request is handled by the authoritative DNS server (ADNS) of the Web site that can therefore serve as a request dispatcher.

Dispatching algorithms implemented at the DNS level have to address some challenging issues. The main problems for load sharing come from the non-uniform distribution of the load among the client domains [14, 91] and from address caching mechanisms that let the authoritative DNS server control only a very small fraction (often on the order of a few percentage) of the requests reaching the Web site. The ADNS specifies the TTL interval, or the period of validity of cached addresses, which is typically set equal for all address mapping requests reaching the ADNS. The limited control of ADNS prevents risks of bottleneck in DNS-based distributed Web-server systems, however it creates a challenge to DNS-based dispatching algorithms. The problem is to find algorithms able to share the load in a system when the control is limited to a few percentage of the total arrivals. This subject is quite new to the existing literature on centralized dispatchers of traditional parallel/distributed systems which have full control on job requests [41, 60, 95, 152].

In this chapter, we will consider the Web-server system as a collection of heterogeneous nodes, each of them having a visible IP address. One node may consist of a single Web server or of multiple servers as in Web clusters. It was shown that,
under realistic scenarios, the application of classical dispatching algorithms, such as round-robin and least-loaded-server, to the ADNS often results in overloaded Web nodes well before the saturation of the overall system capacity [48]. Other dispatching policies, that integrate some client information with feedback alarms from highly loaded Web nodes, achieve much better performance in a homogeneous Web-server system.

The problem of the DNS assignment is complicated by the presence of Web nodes having different capacities. Web systems with so-called heterogeneous nodes are quite likely to be found in reality. Given the non-uniform distribution of the client requests, the limited control of the dispatcher and the node heterogeneity, the ADNS has to make global dispatching decisions under wide uncertainty. This chapter shows that a simple extension of the algorithms for homogeneous Web-server systems proposed in [48] does not perform well in a heterogeneous system. These policies have poor performance even at low levels of node heterogeneity. Other static and dynamic load sharing policies for heterogeneous parallel systems, like those proposed in [5, 107], cannot be used because of the peculiarities of DNS dispatching.

Therefore, in this chapter, we will propose and evaluate new DNS dispatching policies, called adaptive TTL algorithms. Unlike in conventional DNS algorithms where a fixed TTL value is used for all address mapping requests, the tailoring of the TTL value adaptively for each address request opens up a new dimension to perform load sharing. Extensive simulation results show that these strategies are able to avoid nodes overloading very effectively even for high levels of node heterogeneity.

To make both constant and adaptive TTL algorithms applicable to a DNS-based architecture working in a realistic environment, the only requirement is an estimate of state information needed by the DNS dispatching policies with low computation and communication overhead. Thus, a further contribution of this chapter is the analysis of the feasible sources and types of information that can be used to estimate the state information needed in DNS dispatching policies.

The outline of this chapter is as follows. In Section 5.1, a general description of the system environment is provided and relevant state information required to facilitate DNS scheduling are reviewed. In addition, some terminology is introduced also for use in the next chapters. In Section 5.2, we describe various DNS dispatching algorithms with constant TTL, while in Section 5.3 we propose the new class of adaptive TTL algorithms. In Section 5.4, we investigate whether and how system state information used by DNS dispatching policies can be obtained at the ADNS. In Section 5.5, we de-
scribe the model and parameters of the heterogeneous distributed Web-server system used for the performance study. In Section 5.6, we first discuss the appropriate metrics to compare the performance of the algorithms and then present the performance results of the various algorithms for a wide set of scenarios. Section 5.7 contains our concluding remarks.

5.1 DNS-based Web System

In this section, we describe the system environment first. Then, we examine the system state information that can be useful to DNS-based dispatching.

5.1.1 System Environment

In this chapter, we consider the Web-server system as a collection of $N$ heterogeneous nodes $\{S_1, \ldots, S_N\}$, numbered in non-increasing order of processing capacity. Each node may consist of a single server or a Web cluster, it may be based on single- or multi-processors machines, have different disk speeds and various internal architectures. However, from our point of view, we only address heterogeneity through the notion that each node may have a different capacity to satisfy client requests. In particular, each node $S_i$ is characterized by an absolute capacity $C_i$ that is expressed as requests per second it can satisfy, and a relative capacity $\xi_i$ which is the ratio between its capacity and the capacity of the most powerful node in the Web-server system, that is $\xi_i = C_i / C_1$ ($\xi = [\xi_1, \ldots, \xi_N]$ is the so-called vector of relative capacities). Moreover, we measure the heterogeneity level of the distributed architecture by the maximum difference between the relative node capacities, that is $\alpha = \xi_1 - \xi_N$. For example, if the absolute capacities of the nodes are $C_1=150$ requests/sec, $C_2=120$ requests/sec, $C_3=100$ requests/sec, and $C_4=75$ requests/sec, the vector of relative capacities is $\xi = [1, 0.8, 0.66, 0.5]$, and $\alpha = 0.5$.

The Web site built upon the distributed Web-server system is visible to users through a single name, while the multiple IP addresses of the Web nodes (that is, servers or clusters) are visible to clients.

On the user side, the clients have a (set of) local name server(s) and are connected to the network through gateways. We will refer to the sub-network behind these local gateways as a *domain*. From the point of view of the authoritative DNS server, the client can be identified only through the domain it belongs to (assuming that name servers perform only iterative DNS queries), while the Web servers can identify each
individual client through its IP address. We will return on this point in Section 5.4.

5.1.2 State Information for DNS Dispatching

One important issue in dealing with the DNS dispatching problem is the kind of state information that can be used in mapping the site name into IP addresses, as discussed in Section 4.3.3. In this section, we briefly review the different types of state information available at the ADNS from the point of view of their effectiveness [48], while we will describe how to obtain them in Section 5.4.

No state information. DNS dispatching policies, such as round-robin and random used in [8, 98], that do not require any state information show very bad performance under realistic scenarios [48]. (See further discussion in Section 5.6.2.)

Server state information. Even a frequent exchange of detailed information about the present and past load conditions of each Web node (like queue lengths, utilization, etc.) is not sufficient to provide assignment decisions that can avoid overloading some Web node while under-utilizing others. This fact lead us to exclude policies such as least-loaded node from further consideration for a heterogeneous Web-server system.

On the other hand, information on overload nodes is useful to avoid that ADNS assigns address requests to already over-utilized nodes. This can be accomplished through a simple alarm feedback mechanism that monitors the actual load of each node and informs the ADNS when any node is over-utilized [48]. We assume that all of the dispatching algorithms, to be discussed next, consider a node as a candidate for receiving requests only if that node is not over-loaded.

Client domain information. An effective dispatching policy has to take into account some client domain information, because any DNS decision on an IP address resolution affects the selected node for the entire TTL interval, during which the site name to IP address mapping is cached in the name servers. Therefore, the ADNS needs to make an adequate prediction about the impact that each address mapping will have on the future load of the nodes. The key goal is to obtain an estimate of the domain load rate \( \lambda_i \), which is the average number of requests per second reaching the Web-server system from the \( i \)-th domain. Multiplying \( \lambda_i \) by TTL, we obtain the hidden load weight \( w_i \), which is
the average number of requests that each domain sends to a Web node during a TTL interval after a new address resolution request has reached the ADNS.

Also, in a heterogeneous Web-server system, the ADNS must take into account the node processing capacity in either making node assignment or fixing the TTL value.

5.2 DNS Algorithms with Constant TTL

Strategies that do not work well in the homogeneous case cannot be expected in general to achieve acceptable results in a heterogeneous node system. Hence, we will consider only the best performing algorithms that seem to be extensible to a heterogeneous environment. Among the several alternatives proposed in [48], the following policies gave the most promising results; we present them in the extended form to take into account different node capacities.

**Two-tier round-robin (RR2).** This is a generalization of the round-robin (RR) algorithm, based on two considerations [48]. First, being the clients not uniformly distributed, the load rate of domains is typically very different from each other. Secondly, the risk of overloading some of the nodes is mostly due to the requests coming from few very popular domains. Therefore, RR2 uses the domain load rate information to partition the domains connected to the Web cluster into two classes: normal domains and hot domains. In particular, RR2 sets a class threshold and evaluates the relative domain load rate, with respect to the total number of requests from all domains. The domains having a relative load larger than the class threshold belong to the hot class. By default, we set the class threshold to \(1/|\text{domain}|\), where \(|\text{domain}|\) is the average number of domains connected to the nodes. The RR2 strategy applies a round-robin policy to each class of domains separately. The objective is to reduce the probability that the hot domains are assigned too frequently to the same servers. Partitions of domains in more than two classes have been investigated also, but little performance improvement was found.

RR2 (and also RR) can easily be extended to a heterogeneous Web-server system through the addition of some probabilistic assignment features. The concept is to make the RR assignment probabilistic on the basis of the node capacity, as explained in Section 3.3.2. These probabilistic versions of the RR and RR2 algo-
rithms are denoted by Probabilistic-RR (PRR) and Probabilistic-RR2 (PRR2), respectively. Hereafter, we will refer to the conventional RR as Deterministic-RR (DRR) to distinguish it from PRR and similarly, DRR2 from PRR2.

**Dynamically Accumulated Load (DAL).** This algorithm uses the domain load rate to estimate the hidden load weight of each domain [48]. Each time the ADNS makes a node selection following an address request, it accumulates the hidden load weight of the requesting domain in a bin for each node, to predict how many requests will arrive to the chosen node due to this mapping. At each address request, the ADNS selects the node that has the lowest bin level. DAL makes the node selection only based on the hidden load weight from the clients; a generalization of this algorithm to a heterogeneous Web-server system should take into account the node capacity. The solution is to normalize the hidden load weight accumulated at each bin by the capacity of the corresponding node. On IP address assignment, so DAL selects the node that results with the lowest bin level after the assignment.

**Minimum Residual Load (MRL).** This algorithm is a variation of the basic DAL [48]. Like the previous algorithm, MRL tracks the hidden load weight of each domain, but, in addition, the ADNS maintains an assignment table containing all domain to node assignments and their times of occurrence.

Let \( t_j \) be the session average length for a client of the \( j \)-th domain. After a period of \( \text{TTL} + l_j \), the effect of the assignment is expected to expire, that is no more requests will be sent from the \( j \)-th domain due to this assignment. Hence, the entry for that assignment can be deleted from the assignment table. On the arrival of an address resolution request at the time \( t_{\text{now}} \), the ADNS evaluates the expected number of residual requests that each node should receive, on the basis of the previous assignments, and chooses the node with the minimum number of residual requests, that is

\[
\min_{i=1,\ldots,N} \left\{ \sum_{d_j \rightarrow n_i} \sum_k \left[ \frac{(w_j/\xi_i) (t_j(i,k) + \text{TTL} + l_j - t_{\text{now}})_+}{(\text{TTL} + l_j)} \right] \right\}
\]

where \( w_j \) is the hidden load weight of domain \( d_j \), \( \xi_i \) is the relative capacity of node \( n_i \), and \( t_j(i,k) \) is the time of the assignment of the \( k \)-th address resolution
Section 5.3. DNS Algorithms with Adaptive TTL

request coming from domain \(d_j\) to node \(n_i\) in the mapping table. The \((x)_+\) notation denotes that only the positive terms are considered in the internal sum because no more residual load is expected to remain from an assignment when the corresponding term is detected to be negative. The term \((t_j(i, k) + TTL + l_j)\) represents the time instant that the address mapping expires, while the term \((t_j(i, k) + TTL + l_j - t_{\text{now}})_+\) represents the remaining time that the mapping is still valid. By normalizing \(w_j\) by \(\xi_i\), the effect of the node heterogeneity is captured. Unlike the previous two policies that require domain load rate information only, MRL needs also an estimation of the session average length to evaluate the residual load.

5.3 DNS Algorithms with Adaptive TTL

As we shall see in Section 5.6.2, dispatching policies for a homogeneous Web-server system are inadequate to address both node heterogeneity, non-uniform domain load rates, and limited DNS control, even if they are modified for a heterogeneous system. Their poor performance motivated us to search for an alternative approach. Looking at DNS activities, one observes that when an address request reaches the ADNS, the dispatcher returns not only the IP address of the chosen Web node, but also the period (TTL interval) during which this address mapping is valid. Hence, the idea to adaptively adjust the TTL value to reduce the load skew; the proposed class of dispatching algorithms explores the validity of such a criterion.

Adaptive TTL algorithms need information on the processing capacity of each node and the load rate of each connected domain in order to set the TTL value for each address request. Since these are also used by DNS dispatching policies discussed in the previous section, no new information is required to dynamically fix the TTL value.

The basic idea is to deal with the non-uniform distributed domain load rates and/or the heterogeneous node capacities by assigning a different TTL value to each address request. The rationale of such an approach lies in the observation that the hidden load weight increases with the TTL value, irrespective of the domain. Therefore, by selecting an appropriate TTL value for each address resolution request, we can reduce the load skews that are the main cause of overloading, especially in a heterogeneous system. More specifically, we can make subsequent requests from each domain consume similar percentages of node capacity.
TTL selection can address both node heterogeneity and non-uniform load rates. Let us consider node heterogeneity first. A higher TTL value is assigned when the ADNS chooses a more powerful node, while a lower TTL value is set when the requests are assigned to a less capable node. This is because, for the same fraction of node capacity, the more powerful node can handle a larger number of requests, or take requests for a longer TTL interval.

A similar approach can be applied to handle non-uniform load rate distribution: address requests coming from hot domains will receive a lower TTL value than requests originating from normal domains. As the hot domains have higher load rates, a shorter TTL interval will even up the total number of subsequent requests generated.

The new class of dispatching disciplines that apply this approach is named adaptive TTL. It consists of a two-step decision process: in the first step, the ADNS selects the Web node; in the second step, it chooses the appropriate value for the TTL interval. These strategies can be combined with any dispatching algorithm described in Section 5.2, but, we consider the basic RR algorithm and its RR2 variant only. We also combine the adaptive TTL policies with the deterministic and probabilistic versions of these algorithms. They both handle non-uniform requests by using TTL values inversely proportional to the domain load rate, while system heterogeneity is addressed either during the node selection (probabilistic policies) or through the use of TTL values proportional to the node capacities (deterministic policies).

### 5.3.1 Probabilistic Algorithms

The probabilistic policies use PRR or PRR2 algorithms to select the node. Then, the TTL value is assigned according to the load rate of the domain that has originated the address request. In its most generic form, we denote \( \text{TTL}/i \) the policy that partitions the domains into \( i \) classes based on the relative domain load rate and assigns a different TTL value to address requests originating from different domain classes. \( \text{TTL}/i \) is a meta-algorithm that includes various strategies. For \( i = 1 \), we obtain a degenerate policy (TTL/1) that uses the same TTL for any domain, hence not a truly adaptive TTL algorithm. For \( i = 2 \), we have the policy (TTL/2) that partitions the domains into normal and hot domains, and chooses a high TTL value for requests coming from normal domains, and a low TTL value for requests coming from hot domains. Similarly, for \( i = 3 \), we have a strategy that uses a three-tier partition of the domains, and so on, until for \( i = K \) we have the algorithm (TTL/K) that uses a different TTL value for each connected domain. (In actual implementation, domains with lower
load rates may be lumped into one class to reduce the amount of bookkeeping.) For TTL/K policies, if $TTL_j(t)$ denotes the TTL value chosen for the requests coming from the $j$-th domain at time $t$, it will be:

$$TTL_j(t) = \frac{\lambda_{\text{max}}(t) \eta_p}{\lambda_j(t)}$$

(5.1)

where $\eta_p$ is the parameter which scales the average (and the minimum) TTL value and hence the overall rate of the address mapping requests, $\lambda_j(t)$ and $\lambda_{\text{max}}(t)$ are the load rates (estimated at time $t$) of the $j$-th domain and of the most popular domain, respectively.

5.3.2 Deterministic Algorithms

Under the deterministic algorithms, the node selection is done by the ADNS through the deterministic RR or RR2 policy. The approach in handling non-uniform load rates is to adjust TTL value as seen for the probabilistic disciplines. However, the choice of TTL value is now based on the node capacity as well. For the generic TTL/S_i policy, we partition the client domains into $i$ classes depending on their load rates. The TTL for each class and node is set inversely proportional to the class load rate while it is directly proportional to the node capacity. The deterministic TTL/S_1 algorithm is a degenerate case that considers node heterogeneity only and ignores the skew on domain load rates.

The TTL/S_2 policy uses two TTL values for each node, and the selection of either depends on the domain class of the requests, which is normal or hot domain.

The TTL/S_K algorithm selects a TTL value for each node and domain combination. Specifically, $TTL_{ij}(t)$, the TTL chosen for the requests from the $j$-th domain to the $i$-th node at time $t$, is:

$$TTL_{ij}(t) = \frac{\lambda_{\text{max}}(t) \eta_d \xi_i}{\lambda_j(t)}$$

(5.2)

where $\eta_d$ is the parameter which scales the average TTL (and the minimum TTL) value.

5.4 Information Gathering Mechanisms

Most of DNS-based dispatching algorithms considered in this and the following chapters use some state information on server and/or domain. In particular, the algorithms
described in the previous sections base their decision on the feedback alarm mechanism coming from overloaded Web nodes and on the estimated load rate of each client domain connected to the Web site. A major issue is whether these pieces of information are actually accessible to the authoritative DNS server of the distributed Web-server system. We assume as first requirement for a DNS-based system that it is to be fully compatible with the existing Web standards and protocols. Therefore, since we cannot force any Web entity to collect information for the ADNS dispatcher, we must have any necessary state information collected on the ADNS by the system nodes and the ADNS itself; these only are the entities that the Web site management can use to collect and exchange load information. Algorithms and mechanisms that need active cooperation from any other Web entity, such as Web browsers, name servers, and users, are beyond the scope of this thesis, because they require modifications of some out-of-control Web components. We examine next how the ADNS can have access to the feedback alarm and the domain load rate information.

5.4.1 Server State Information

The implementation of a feedback alarm information requires two simple mechanisms: a monitor of the load for each Web node, and an asynchronous communication protocol between the nodes and the authoritative DNS server. Each node periodically calculates its utilization and checks whether it has exceeded a given \( \vartheta \) load threshold (typically set to 10-15% over the average system utilization). If affirmative, the node sends an alarm signal to the ADNS, which will exclude it from any further assignment until its load falls below the threshold. This last event is communicated to the ADNS through a normal signal. If every node in the system generates an alarm and the table of available nodes is empty, the ADNS replies to IP address requests that the requested mapping is not available. However, this is a rare event, which occurred very few times in our experiments for all proposed policies.

Although fault-tolerance is not the focus of this thesis, it is worth noting that a very simple modification of this feedback mechanism could avoid requests being routed to failed or unreachable nodes. Either node-initiated (through synchronous messages) or dispatcher-initiated (through a polling mechanism) strategies could be combined with the dispatching algorithm to provide fault-tolerance.

The node processing capacity information \( \{\xi_i\} \) is a static information. It is an estimate of the number of hits, or HTTP requests, per second that each node can support.
5.4.2 Domain State Information

The estimation of the domain load rate cannot be done by the ADNS alone because the information coming from clients to the ADNS is very limited. For each client session requiring an address resolution, the ADNS sees only the IP address of the local name server for the client’s domain (and not the IP address of the client requesting to the Web site). Due to address caching, the ADNS will see a new address request coming from the same domain only after TTL seconds, regardless of the domain load rate. Hence, the only viable approach to estimate the load rate information requires cooperation of the Web nodes. They can track and collect the load offered to the distributed Web-server system through the access logfile that each Web server maintains to trace the client accesses in terms of HTTP requests. According to the Common Logfile Format [49], the information for each request include the remote hostname (or IP address), the requested URL, the date and time of the request, and the request type. Extended logs also provide referred information for linking each request to a previous Web page request from the same client.

There are two relevant issues to discuss about the estimation of the domain load rate:

- which is the granularity of the information that can be extracted from the servers’ logfile and which is the most useful;
- how logged requests at the Web server can be grouped together into a client domain and also how the domain can be matched with the IP address of the local name server seen by the ADNS.

Information for Domain Load Rate Estimation

In this section, we assume that each node is able to identify a client domain from the IP address of the requesting client (supposing that the logfiles contain only IP addresses); we will look at the methods for identifying a domain in the next section.

Each node periodically sends its estimate of the domain load rates to the ADNS, where a collector process gets all estimates and computes the actual load rate from each domain by adding up the load rate of it on each node. The domain load information can be extracted from the logfile in connection with different granularity of details: either the number of sessions, the number of page requests or the number of hits (or Web objects). The domain load rate, when estimated through one of the
aforesaid information, is referred to as domain session rate, domain page rate, and domain hit rate, respectively.

The number of sessions from each domain gives just a rough approximation of the domain load rate. Moreover, getting this information may not be as straightforward as one may think, since HTTP protocol is stateless (see Section 2.3). Session information is more relevant to the MRL policy, where an estimate of the session average length is required in addition to the domain load rate. Some heuristic methods for session identification are described in [11, 35, 105, 106].

The number of page requests from each domain provides an alternative approximation of the domain load rate. This value can be measured for each domain by excluding non-HTML requests (e.g., gif files) from the counting.

Estimating the workload as a function of the number of hits from each domain is the most accurate way to assess the domain load rate. Another advantage of this information is to be the least expensive as to get it only a sequential scan of the logfile is required. In so doing, each entry in the logfile coming from a certain domain is counted as hit for that domain.

We have conducted extensive simulations to demonstrate that the proposed DNS dispatching policies, when integrated with some heuristics for domain load estimation, are still effective, even for highly skewed load in both static and dynamic scenario (the latter being the case in which the domain load rates undergo dynamic variations [35]). In particular, the approach that estimates the domain load rate through the hit information, combining more samples through a weighted mean, appeared to give more stable results. Additional details can be found in [35].

**Clustering of Clients into Domains**

When handling a site name-to-address request, the authoritative DNS server of the Web site typically sees as the originator only the IP address of the client’s local name server. It has no way of knowing who the actual client is. On the other hand, Web server logs contain the client IP address. Therefore, it is necessary to define a method to group clients seen by the Web servers into domains (we use the term clustering to denote such a grouping), as well as a method to match these domains with the corresponding local name servers which send DNS queries to the authoritative DNS server.

A simple approach to identify client domains is to assume that all clients sharing the same mask (referred to as domain mask) in their IP addresses belong to the same
domain. The authoritative name server can use the same heuristic to identify the domain, which the requesting local name server belongs to. The domain mask can be either the first 24 bits of the IP address, or the network prefix based on class A, class B, and class C networks. As an example, let us consider the following client IP addresses in the server logfile: 160.80.85.30, 160.80.85.37, and 160.80.1.5. According to the first method, 160.80.85 and 160.80.85.37 are clustered into the same domain being 160.80.85 their common mask, while 160.80.1.5 belongs to a different domain, with mask 160.80.1. On the other hand, if the latter method applies, all clients are grouped into the same domain, as 160 is a network prefix of class B.

Both heuristics have some drawbacks. Identifying client domains on the first three bytes of the IP address has two disadvantages: first, it mis-identifies small clusters; secondly, it may mis-cluster all class A, class B, and CIDR networks\(^\text{1}\). Similarly, identification through the network prefix can also be inaccurate, due to subnetting within class A and class B as well as CIDR addressing. On the positive side, even if they do not provide an accurate clustering of clients into domains, still these methods can be applied in DNS dispatching effectively. Indeed, the main concern is to identify popular domains, as they are the ones that heavily impact on the performance of DNS-based Web systems. Furthermore, the above heuristics do not introduce a significant overhead for client clustering on both Web servers and ADNS.

Other heuristic methods of grouping clients into domains are based either on Autonomous System (AS) numbers (that is, clients sharing the same AS number belong to the same domain) or on domain names [138]. Also these methods are not problem-free. The use of AS numbers requires to obtain a local copy of the Internet Routing Registry (IRR) databases for AS number lookups. The use of domain names requires to perform a reverse address resolution to map the IP addresses found in the logfile into the corresponding hostnames. Since a one-to-one mapping is not available for all names (e.g., due to the use of dynamic IP addresses), this approach is both time consuming (as it requires a reverse lookup) and imprecise (in [91], the authors were unable to obtain names for about 50% of the addresses).

More accurate and time consuming client clustering techniques are based on routing information, like those provided by BGP routing tables [91]. It is worth noting that the DNS protocol modification proposed in [138], which addresses the problem

---

\(^1\)Classless Inter-Domain Routing (CIDR) has been proposed to address the possible exhaustion of class B network address space by collapsing a block of contiguous class C addresses into a single entity [97].
of identifying clients during the address lookup phase, is useful also for the purpose of estimating domain load rates. As in [138] the authors propose to extend the standard DNS message by including also the IP address of the actual client requesting name resolution, the client IP address can be used by the ADNS to identify easily which domain the client belongs to using the same information seen by the Web nodes.

To conclude this section, we present a description of the architecture of the DNS-based system as outlined in this chapter. Figure 5.1 contains all the software components needed for ADNS, assuming that a node consists of a single Web server machine. Besides the DNS base function, the authoritative DNS server includes a dispatcher, an alarm monitor, a domain load collector, and a TTL selector. The ADNS dispatcher assigns each address request to one of the nodes based on some algorithm. The alarm monitor tracks the feedback alarm from servers to avoid assigning requests to an overloaded node until its load level is returned to normal. Instead, the domain load collector gathers load information from each node and estimates the domain load rate (and, when necessary, the hidden load weight) of each connected domain. The TTL selector sets the appropriate TTL value for the address mapping when adaptive TTL algorithms are employed.

Also shown are the corresponding components in the Web node; besides the HTTP daemon server, these include a load monitor and a request counter. The load monitor tracks the node load and issues alarm or normal signal according to the procedure explained above. The request counter estimates the number of requests received from each domain in a given period of time, and periodically provides the information to the domain load collector in ADNS. When the ADNS request collector does not receive any message from the request counter within a certain time, it assumes that the node is either unreachable, faulty or overloaded. If so, the dispatcher excludes this node from the table of available servers as though an alarm signal were received by the alarm monitor.

When the node is a Web cluster with multiple server machines, the request counter and load monitor processes can run on the Web switch. This component would have the twofold role of intra-cluster information collector and interface with the ADNS.

5.5 Simulation Model

In this section, the simulation model and its various parameters are described.
First, let us have a look at the workload. We assume that clients are partitioned among the domains based on a Zipf's distribution, that is a distribution where the probability of selecting the $i$-th domain is proportional to $1/i^{(1-x)}$ \cite{Zipf}. This choice is motivated by several studies demonstrating that if one ranks the popularity of client domains by the frequency of their accesses to the Web site, the distribution of the number of clients in each domain is a function with a short head (corresponding to big providers, organizations and companies, possibly behind firewalls), and a very long tail. In our experiments, the clients are partitioned among the domains based on a pure Zipf's distribution, that is using $x = 0$, in the default case. This represents the most skewed client distribution. Additional sensitivity analysis on the skew parameter ($x$) is included in Section 5.6.5.

We did not model the details of Internet traffic \cite{Rogers, Broido} as this chapter is focused on Web-server system performance. However, we have considered the main components that affect the performance of the system. These include an accurate representation of both number and distribution of the intermediate name servers as in \cite{Deola}, which influence action and performance of the DNS dispatching algorithms through their address caching mechanisms.

Moreover, we consider all the details concerning a client session that is the total amount of time during which a single user accesses to the Web site. In the first step, the client obtains (through the ADNS or the cache of a name server) an address
mapping to one of the Web nodes through the address resolution process. As the Web-server system consists of heterogeneous nodes with identical content, the requests from the clients can be assigned to any of the nodes. Once the node selection has been completed, the client is modeled to submit multiple Web page requests that are separated by a given user think time. The number of page requests per session and the time between two page requests from the same client are assumed to be exponentially distributed as in [14].

Each page request consists of a burst of small requests sent to the node. These bursts represent the objects that are contained within a Web page. Under the HTTP/1.0 protocol, each object is requested on a new connection between the client and the Web node. However, address caching at the browser level guarantees that a client session is served by the same Web node independently of its duration. Moreover, the new version HTTP/1.1 provides persistent connections during the same session [65]. As a consequence, the differences between different versions of the HTTP protocols do not affect the results achieved in this chapter.

The number of Web objects per page is obtained from a uniform distribution in the discrete interval [5-15]. The object service time and the inter-arrival time of object requests to the node are assumed to be exponentially distributed. (In Section 5.6.5, we will also consider the case of objects with very long service time to resemble dynamic Web content.) Other parameters used in the experiments are reported in Table 5.5, with their default values between brackets (time values are in seconds). When not otherwise specified, all performance results refer to the default values. A thorough sensitivity analysis herein not shown revealed that the main conclusions of the experiments are not affected by the choice of workload parameters such as the number of Web objects per page request, the mean service time and inter-arrival time of objects.

In our experiments, we considered five levels of node heterogeneity. Table 5.5 reports details about the relative capacities and the heterogeneity level of the Web-server system. By carefully choosing the workload and system parameters, the average system utilization is kept to 2/3 of the whole capacity. This value is obtained as a ratio between the offered load, that is the total number of requests per second arriving to the Web site, and the system capacity which is the sum of each node capacity denoted in requests per second. While we considered different levels of node heterogeneity, we keep the system capacity constant for a fair performance comparison among the proposed algorithms.
Table 5.1: Parameters of the system model.

<table>
<thead>
<tr>
<th>Category</th>
<th>Parameter</th>
<th>Value (default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web system</td>
<td>Number of nodes</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>System capacity</td>
<td>1500 req/sec</td>
</tr>
<tr>
<td></td>
<td>Average system load</td>
<td>1000 req/sec</td>
</tr>
<tr>
<td></td>
<td>Homogeneous $\alpha_A = 0$</td>
<td>$\xi_A = [1, 1, 1, 1, 1, 1, 1]$</td>
</tr>
<tr>
<td></td>
<td>Heterogeneity $\alpha_B = 0.2$</td>
<td>$\xi_B = [1, 1, 0.8, 0.8, 0.8, 0.8, 0.8]$</td>
</tr>
<tr>
<td></td>
<td>Heterogeneity $\alpha_C = 0.35$</td>
<td>$\xi_C = [1, 1, 0.8, 0.8, 0.65, 0.65, 0.65]$</td>
</tr>
<tr>
<td></td>
<td>Heterogeneity $\alpha_D = 0.5$</td>
<td>$\xi_D = [1, 1, 0.8, 0.8, 0.5, 0.5, 0.5]$</td>
</tr>
<tr>
<td></td>
<td>Heterogeneity $\alpha_E = 0.65$</td>
<td>$\xi_E = [1, 1, 0.8, 0.8, 0.35, 0.35, 0.35]$</td>
</tr>
<tr>
<td>Domain</td>
<td>Connected</td>
<td>10-100 (20)</td>
</tr>
<tr>
<td></td>
<td>TTL (constant)</td>
<td>0-700 (240)</td>
</tr>
<tr>
<td></td>
<td>TTL (adaptive)</td>
<td>0-2400</td>
</tr>
<tr>
<td>Client</td>
<td>Number</td>
<td>1000-3000 (1500)</td>
</tr>
<tr>
<td></td>
<td>Distribution among domains</td>
<td>Zipf $(x = 0, 0.5, 1)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Geometric $(p = 0.3)$</td>
</tr>
<tr>
<td></td>
<td>Web page requests per session</td>
<td>exponential (mean 20)</td>
</tr>
<tr>
<td></td>
<td>Objects per Web page</td>
<td>uniform in [5-15]</td>
</tr>
<tr>
<td></td>
<td>User think time</td>
<td>exponential (mean 15)</td>
</tr>
<tr>
<td></td>
<td>Inter-arrival time of objects</td>
<td>exponential (mean 0.25)</td>
</tr>
<tr>
<td></td>
<td>Object service time</td>
<td>exponential (mean $1/C_i$)</td>
</tr>
</tbody>
</table>

To implement the feedback alarm, we have each node periodically calculate its utilization (with period set to 16 seconds) and check whether it has exceeded a given $\theta = 0.75$ threshold as in [48]. (Additional simulations, not shown here, indicate that our results are not sensitive to these parameters.)

For the constant TTL schemes, a default TTL value of 240 seconds is used as in [57]. For the adaptive TTL algorithms, the average TTL value is fixed to 400 seconds so as to keep the minimum TTL value (henceforth, denoted as $TTL$) above 60 seconds, while the maximum TTL value can go up to 1200 seconds. This average TTL value is considerable higher than the 240 seconds in the constant TTL case. Nonetheless, as we shall see later, the adaptive TTL algorithms still perform far better than the constant TTL algorithms. Sensitivity analysis to TTL values is provided in Section 5.6.5.

The simulator, based on the Independent Replication Method [99], was implemented using the CSIM18 package [108]. Each value is the result of five or more simulation runs with different seeds, where each run is made up of five hours of the Web site activities. Confidence intervals were estimated on all simulation results, and the 95% confidence interval was observed to be within 4% of the mean.
5.6 Experimental Results

In this section, we first discuss the metrics to compare the performance of different DNS algorithms. Then, we present the simulation results. For the performance evaluation of the proposed dispatching algorithms, a large number of experiments was carried out. The first set of experiments in Section 5.6.2 shows which the problem with constant TTL algorithms is. It highlights the point that just considering the dispatching component is not sufficient to achieve good performance. Then, we evaluate algorithms that also explore the TTL component in Section 5.6.3. The remaining sections focus on measuring how effectively adaptive TTL algorithms applied to an ADNS dispatcher, which is characterized by a limited control on assignment, can avoid overloading nodes in a heterogeneous Web-server system.

5.6.1 Performance Metrics

In this section, we examine the metrics apt to evaluate the performance of a DNS algorithm in a heterogeneous Web-server system. Our main goal is to avoid any of the Web nodes becoming overloaded, that is to say, our objective is to minimize the highest load among all nodes at any instant. Commonly adopted metrics such as the standard deviation of node utilization are not useful to the purpose, because minimizing the load differences among the Web nodes hits only a secondary goal.

Therefore, we will evaluate the performance of the various policies focusing on the system maximum utilization at a given instant, that is the highest node utilization observed among all nodes in the system at that instant. As an example, let us assume three nodes in a Web-server system. If their utilization is 0.6, 0.75, and 0.63, respectively, at time $t_1$, and 0.93, 0.66 and 0.42, respectively, at time $t_2$, the system maximum utilization at $t_1$ is 0.75 and that at $t_2$ is 0.93. With a system maximum utilization of 0.93, the Web site has serious load problems at $t_2$.

Specifically, the major performance criterion is the cumulative frequency of the system maximum utilization, or the probability (or fraction of time) that the system maximum utilization is below a certain value. By looking at the highest utilization among all Web nodes, we can deduce whether the Web-server system is overloaded or not. Moreover, its cumulative frequency can provide an indication on the relative frequency of overloading. For example, if the probability that all nodes are utilized less than 0.80 is 0.75, then the probability of at least one node exceeding 0.80 is 0.25.

In practice, the performance of the various dispatching policies is evaluated by
Section 5.6. Experimental Results

tracking the system maximum utilizations periodically during the simulation runs. The node with the maximum utilization changes over time; however, if the system maximum utilization at an instant is low, no node is overloaded at that time. By tracking the time when the system maximum utilization is above or below a certain threshold, we get to know how well the Web-server system is working. In our experiments the Web-server system is subject to an offered load equal to 2/3 of the overall system capacity, but typically all the nodes, even if with different proportions, contribute to this maximum during an entire simulation run. Since the average utilization is fixed at 2/3, the distribution of the system maximum utilization of a perfect policy (always maintaining a utilization of 2/3 at each node) should be a step function which goes from 0 to 1 at a utilization of 2/3.

When the sensitivity of the dispatching algorithms is evaluated as a function of system parameters, such as node heterogeneity, we find useful to adopt a metric linked to the cumulative frequency of the system maximum utilization. For this reason, we consider the 96-percentile of the system maximum utilization, that is \( P[\text{System Maximum Utilization} < 0.96] \). In other words, the probability of no node of the Web-server system being overloaded, namely \( \text{Prob(Not Overloaded System)} \), becomes the performance metric of interest.

5.6.2 Performance of Constant TTL Algorithms

In this section we evaluate the performance of the constant TTL algorithms based on the parameters of Table 5.5. We ran simulations for four heterogeneity levels of the Web-server system ranging from 20% to 65%. In Figure 5.2, we present the performance results for the lowest heterogeneity level. In this figure, we also report as "ideal" policy the PRR algorithm under uniform distribution of the client request rates, and the random algorithm under Zipf distribution, which has the worst performance. The system maximum utilization is on the \( x \)-axis, while the \( y \)-axis is its cumulative probability (or relative frequency). The higher the probability the less likely that some of the nodes will be overloaded, and hence better load sharing is achieved. For example, under the PRR2 policy, the probability of having maximum utilization below 0.95 is about 0.5, while under DRR, the probability is only 0.2. This figure confirms that the various probabilistic versions of the round-robin policy perform better than the deterministic versions, and the improvement is even more remarkable for the RR2 algorithm. MRL policy gives results close to PRR2 but much better than the DRR algorithm which is often proposed for DNS-based distributed systems. Indeed, for the
last policy, the probability that no node is overloaded is below 0.2, that is to say, for more than 80% of the observed time there is one overloaded node at least.

However, even considering a slightly heterogeneous system, no strategy achieves acceptable performance. In the best instance, the Web-server system has at least one node overloaded ($P[\text{System Maximum Utilization} < 0.96]$) for about 30% of the time, and the shapes of all cumulative frequencies are very far from the “ideal” policy’s behavior.

![Figure 5.2: Performance of constant TTL algorithms (Heterogeneity level of 20%).](image)

![Figure 5.3: Sensitivity to system heterogeneity of constant TTL algorithms.](image)

Figure 5.3 analyzes the sensitivity of constant TTL algorithms to system heterogeneity. Now on the $y$-axis is the probability that no Web node is overloaded, while on the $x$-axis is the heterogeneity level of the Web-server system. The figure shows that both DAL and MRL-based policies are unable to control node load when the distributed Web-server system is heterogeneous. Although the PRR2 algorithm often performs better than other policies, no strategy clearly outperforms all the others for all system heterogeneity levels. The difficulty of determining the best strategy is also confirmed by other (not reported) experiments in which the number of domains, clients and average load have been changed. Summing up, none of these policies can either be considered adequate, because the probability of having at least one overloaded node is high and always over 0.30.

### 5.6.3 Comparison of Constant and Dynamic TTL Algorithms

The next set of results evaluates to what extent system heterogeneity affects the performance of the adaptive TTL algorithms. Figure 5.4 compares various deterministic
Section 5.6. Experimental Results

TTL/S\_i algorithms for a low heterogeneity level of 20\%. Each set consists of both the deterministic RR2 and RR dispatching schemes for \( i = 1, 2 \) and \( K \). Also shown in this figure is the DRR scheme with a constant TTL. First of all, one can observe that for any set of TTL strategies, the RR2 dispatching scheme is always slightly better than the RR scheme. All adaptive TTL schemes perform significantly better than constant TTL policies when they address both node and client heterogeneity; while their performance does not improve significantly when taking into account only the node heterogeneity (as in TTL/S\_1 schemes). Moreover, the results of these strategies that use a different TTL for each node and domain, namely DRR-TTL/S\_K and DRR2-TTL/S\_K, are very close to the envelope curve of the “ideal” PRR(uniform) policy.

![Figure 5.4: Performance of deterministic algorithms (Heterogeneity level of 20\%).](image1)

![Figure 5.5: Performance of probabilistic algorithms (Heterogeneity level of 20\%).](image2)

Similar results are achieved by the probabilistic schemes that combine adaptive TTL to handle non-uniform domain load rates with probabilistic routing features to address system heterogeneity. Figure 5.5 shows the cumulative probability of the system maximum utilization for a heterogeneity level of 20\%. The relative ranking among the strategies remains unchanged vs. the previous one. Specifically, the RR2 dispatching policies are slightly better than RR strategies and TTL/K strategies outperform TTL/2 strategies. Furthermore, all probabilistic adaptive TTL approaches are considerably better than the PRR scheme with a constant TTL, even for a heterogeneity level as low as 20\%.

Being the RR2-based algorithms better performing than RR-based, we will focus on them in the remainder of this section. Figures 5.6 and 5.7 refer to a Web-server system with a 35\% and 65\% heterogeneity level, respectively. Figure 5.6 shows that
when we adopt a TTL proportional to each domain load rate, the deterministic strategy TTL/S_K prevails over the probabilistic TTL/K. On the other hand, when we consider only two classes of domains, the probabilistic approach TTL/2 is slightly better than the deterministic TTL/S_2. Analogous results are observed for a system heterogeneity equal to 50% and for other system parameters.

<table>
<thead>
<tr>
<th>System Maximum Utilization</th>
<th>Cumulative Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6</td>
<td>0</td>
</tr>
<tr>
<td>0.7</td>
<td>0.2</td>
</tr>
<tr>
<td>0.8</td>
<td>0.4</td>
</tr>
<tr>
<td>0.9</td>
<td>0.6</td>
</tr>
<tr>
<td>1</td>
<td>0.8</td>
</tr>
</tbody>
</table>

**Figure 5.6:** Performance of RR2-based algorithms (Heterogeneity level of 35%).

**Figure 5.7:** Performance of RR2-based algorithms (Heterogeneity level of 65%).

The probabilistic approaches tend to perform better than deterministic strategies when the system heterogeneity is very high, that is over 60%. Figure 5.7 shows that DRR2-TTL/S_K performs best if we look at the 98-percentile, while the shape of the curve is in favor of PRR2-TTL/K if we consider lower percentiles. Moreover, the DRR2-TTL/S_2 and PRR2-TTL/2 algorithms, which performed similarly in all previous cases, differentiate from each other in favor of the probabilistic algorithms. Both DRR2-TTL/S_2 and DRR2-TTL/S_1 seem rather inadequate to address high heterogeneity levels.

We next consider the average number of messages to the ADNS under the constant TTL and adaptive TTL schemes. Specifically, in Figure 5.8, PRR2 and PRR2-TTL/K are chosen to represent the constant TTL and adaptive TTL schemes, respectively. (The difference among the schemes in each class, such as PRR2-TTL/K and DRR2-TTL/S_K, is small.) Figure 5.8 shows both the number of address requests and alarm requests with a 35% heterogeneity level. The adaptive TTL and constant TTL schemes are comparable in terms of request load to the ADNS. It is important that no policy risks to stress the ADNS.
Section 5.6. Experimental Results

5.6.4 Sensitivity to System Heterogeneity

In this set of experiments we evaluate the sensitivity of the proposed strategies for a degree of system heterogeneity ranging from 20% to 65%. We first analyze deterministic and probabilistic policies (Figure 5.9 and Figure 5.10, respectively), and then compare the two classes of policies under RR2 in Figure 5.11. On the $y$-axis is the probability that no node of the Web-server system is overloaded, while on the $x$-axis is the heterogeneity level.

![Figure 5.8: Number of messages to the ADNS.](image)

![Figure 5.9: Sensitivity to system heterogeneity of deterministic algorithms.](image)

![Figure 5.10: Sensitivity to system heterogeneity of probabilistic algorithms.](image)

Figures 5.9-5.11 show that most of the adaptive TTL algorithms are relatively stable that is, their performance does not vary widely when the heterogeneity level increases to 50%. After this level, a more sensible degradation of performance can
be observed for all policies. However, for any heterogeneity level, a large gap exists between the schemes that use a different TTL value for each connected domain and other policies. Moreover, beyond achieving the best performance, the TTL/S\_K and TTL/K algorithms have the best stability, too. TTL/2 algorithms are still acceptable when combined with RR2-based dispatching schemes, while they tend to degrade for higher heterogeneity levels when they are combined with the RR-based dispatching schemes. The DRR-TTL/S\_1 strategy (as shown in Figure 5.9) performs much worse than any other adaptive TTL policy and is rather similar to a constant TTL strategy (comparing Figure 5.9 with Figure 5.3). Hereafter, we will not anymore consider this strategy, because of its instability and poor performance. Figure 5.11 shows that when we assign a different TTL to each connected domain, DRR2-TTL/S\_K performs the best, while with only two classes of domains PRR2-TTL/2 performs better than DRR2-TTL/S\_2 for higher heterogeneity levels.

![Figure 5.11: Sensitivity to system heterogeneity of RR2-based algorithms.](image)

From all the shown results, the following are observed.

- Adaptive TTL schemes, especially DRR2-TTL/S\_K and PRR2-TTL/K, are very effective in avoiding overloading the nodes even when the system is highly heterogeneous and domain load rates are unevenly distributed as in the pure Zipf’s function.

- Constant TTL strategies can handle neither the non-uniform client distribution nor the heterogeneity of nodes.

- Differentiating the requests coming from both popular and normal domains improves the performance, no matter if TTL is dynamically chosen or fixed.
Indeed, RR2-based strategies are always slightly better than their RR-based counterparts.

- Deterministic strategies typically perform better than their probabilistic counterparts. However, the gap is not wide and tends to shrink for high heterogeneity levels.

### 5.6.5 Sensitivity to TTL Values and Workload Parameters

We now consider sensitivity to TTL values. In Figure 5.12, both the PRR2-TTL/K and PRR2 are shown with a 20\% heterogeneity level for average TTL values ranging from 300 to 500 seconds (DRR2-TTL/S_K schemes which behave like PRR2-TTL/K schemes are not shown for readability of the figure). Adaptive TTL schemes outperform remarkably the constant TTL scheme (PRR2) regardless of the TTL values. In Figure 5.13, the DRR2-TTL/S_K and PRR2 are shown with a 50\% heterogeneity level for various mean TTL values (PRR2-TTL/K schemes are not shown for readability of the figure). The superiority of the adaptive TTL schemes is again evident.

![Figure 5.12: Sensitivity to TTL of probabilistic algorithms.](image1.png)

![Figure 5.13: Sensitivity to TTL of determinstic algorithms.](image2.png)

Figure 5.14 shows how sensitive the overload probability is to the mean TTL value for the various TTL policies, when the heterogeneity level is 20\%.

The various adaptive TTL schemes perform far better than the constant TTL ones, PRR and PRR2, and show much lower sensitivity to the TTL values. Among the adaptive TTL schemes, DRR2-TTL/S_K provides the best performance. Also all the RR2 policies perform better than the corresponding RR policies.

We study the sensitivity to the distribution of client requests next. In addition
to the pure Zipf distribution (with the skew parameter $x = 0$), a geometric distribution (with $p = 0.3$), a Zipf distribution with $x = 0.5$, and a uniform distribution (corresponding to $x = 1$) are considered. Figures 5.15 and 5.16 show respectively the probability density function and the cumulative distribution function, for the distribution of clients among 20 domains, ranked by popularity.

Figure 5.15: Probability density function of client distribution.

Figure 5.16: Cumulative distribution function of client distribution.

Figure 5.17 shows the performance of PRR2-TTL/K under the different client distributions with a heterogeneity level of 35%. The performance improves as the skew in the client distribution decreases. The geometric distribution has performance close to the pure Zipf distribution. We note that the mean TTL value is kept at 400 seconds unchanged for all cases. Since the client distributions have different skew, the minimum TTL value under PRR2-TTL/K will be different for the different client
distributions (See Equation 5.1). If we hold the minimum TTL value ($\text{TTL}$) at 60 seconds for all cases as in Figure 5.18, the performance gap gets much larger as the skew in the client distribution increases. The spread of the TTL values among the client domains (hence also the mean TTL value) also increases with the skew as indicated in Figure 5.18.

![Figure 5.17: Sensitivity to client distribution (same mean TTL=400 seconds).](image1)

![Figure 5.18: Sensitivity to client distribution (same $\text{TTL} = 60$ seconds).](image2)

Next, the sensitivity to the object service time is examined. We consider the case where some of the object requests have a particularly long service time, like dynamic requests. We introduce a new type of Web page requests consisting of one object of dynamic type, and five static objects as considered before; the average service time of dynamic objects is supposed to be 10 times the static objects’ one. Figure 5.19 shows the performance of PRR2-TTL/K under different percentages of Web page requests containing a dynamically generated object. The dynamic TTL scheme handles workload with a long object service time very well; its performance actually improves as the percentage of Web page requests containing dynamic objects grows. This is because for a given total load to the system, if the per request load increases, the number of subsequent requests arriving during the TTL period will decrease and therefore the ADNS control improves.

5.6.6 Robustness of Adaptive TTL Algorithms

The previous results point up a clear preference for the adaptive TTL schemes. We now examine their robustness from two specific aspects:

- the impact of non-cooperative name servers, that is not following the TTL value
Figure 5.19: Sensitivity to dynamic page requests of PRR2-TTL/K algorithms.

recommended by the ADNS (see Section 2.2.1);

- the sensitivity of the performance to the accuracy of the estimated domain load rate.

The latter aspect is less of an issue if the load from each domain remains either relatively stable or changes slowly. Instead, in a more dynamic environment where domain load rates may change continuously, it can be difficult to estimate them accurately.

Effects of Non-cooperative Name Servers

To avoid network saturation due to address resolution traffic, very small TTL values can be ignored by some name servers, as explained in Section 2.2.1. Since there is no common TTL lower threshold being adopted by all name servers, in our study we consider the worst case scenario. In it, we consider all name servers as non-cooperative if the TTL proposed by the ADNS is lower than a given minimum, and perform sensitivity analysis for this threshold value.

Figure 5.20 shows the sensitivity of the adaptive TTL policies to the minimum TTL value accepted by the name servers, when the heterogeneity level is 35%. The performance of DRR2-TTL/S_K and PPR2-TTL/K gradually deteriorates as the minimum TTL value allowed by the name servers increases, while PRR2-TTL/2 is almost insensitive to the minimum accepted TTL. The motivation is that the TTL/S_K schemes may sometimes need to select a low TTL value when a client request, coming from a hot domain, is assigned to a node with limited capacity. On the other hand, a
probabilistic TTL/2 strategy is almost insensitive to the presence of non-cooperative name servers because it uses a rough partitioning (just two classes) of the domains and is able to always assign TTL values higher than 180 seconds in the experiments shown in Figure 5.20.

![Figure 5.20: Sensitivity to minimum accepted TTL by name servers.](image)

**Effects of the Estimation Error**

Next, we examine how the maximum error in estimating the load rate of each domain may affect the system performance. Figures 5.21 and 5.22 compare various adaptive TTL schemes in connection with the estimation error for heterogeneity levels of 20% and 50%, respectively. In our experiment, we introduce a perturbation to the load rate of each domain, while the ADNS estimates of the domain load rates remain unchanged. Hence, the percentage error in the load estimate is equal to the percentage of perturbation on the actual load. For the case of a χ% error, the load rate of the busiest domain is increased by χ% while the load rates of the other domains are proportionally decreased so to maintain the total load on the system unchanged. This effectively increases the skew of the load rate distribution, thus representing a worst case.

When the estimation error or load perturbation increases, the system performance decreases for all eight algorithms. However, all the TTL/S_K and TTL/K schemes clustered at the top of the figure show much less sensitivity than the TTL/S_2 and TTL/2 schemes which are at the bottom. In particular, when the node heterogeneity is high (≥ 50%) and the error is large (≥ 30%), the performance of TTL/S_2 strategies can degrade remarkably. Conversely TTL/S_K and TTL/K schemes are
only slightly affected by the error in estimating the domain load rate. Indeed, when the heterogeneity level is less than 50%, their performance degrades at most a few percent as against the case of no estimation error. This proves the robustness of the TTL/S_K and TTL/K algorithms.

Although the TTL/S_2 and TTL/2 algorithms are very sensitive to the estimation error (even when it is as small as 10%), it is important to note that the shown results refer to a positive perturbation induced to the client domain with the heaviest load rate. This is actually a worst (unrealistic) case because the skews of the client request distributions are indirectly increased to exceed a pure Zipf's distribution. Other results not reported, which consider negative perturbations on the heaviest domain load rate, show analogous performance for the TTL/K policies and much improved performance for the TTL/S_2 and TTL/2 strategies. This was expected as a reduction in the load rate of the busiest domain makes client requests more evenly distributed than a pure Zipf’s distribution.

5.6.7 Summary of the Performance Study

Table 5.2 outlines the specific methods that each DNS algorithm uses to address heterogeneous nodes and non-uniform distribution of client requests.

In summary,

- To share the load across multiple Web nodes, the authoritative DNS server has two control knobs: the dispatching policy (that is, the node selection) and the TTL value for the period of validity of the selection.
Section 5.6. Experimental Results

<table>
<thead>
<tr>
<th>Category</th>
<th>DNS Algorithm</th>
<th>Non-uniform Client Distribution</th>
<th>Heterogeneous Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant TTL</td>
<td>DRR</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>DRR2</td>
<td>two-tier domain partition</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>PRR</td>
<td>—</td>
<td>probabilistic routing</td>
</tr>
<tr>
<td></td>
<td>PRR2</td>
<td>two-tier domain partition</td>
<td>probabilistic routing</td>
</tr>
<tr>
<td></td>
<td>DAL</td>
<td>accumulated hidden load weight</td>
<td>normalized bin</td>
</tr>
<tr>
<td></td>
<td>MRT</td>
<td>residual hidden load weight</td>
<td>normalized (residual) bin</td>
</tr>
</tbody>
</table>

| Adaptive TTL (Deterministic) | DRR-TTL/S í | —                                | TTL\(x\) (node capacities) |
|                               | DRR-TTL/S í | TTL\(x\) (i-classes load rate)  | TTL\(x\) (node capacities) |
|                               | DRR2-TTL/S í | two-tier domain partition        | TTL\(x\) (node capacities) |
|                               | DRR2-TTL/S ñ | two-tier domain partition        | TTL\(x\) (node capacities) |

| Adaptive TTL (Probabilistic) | PRR-TTL/í | (same policy as PRR) | probalistic routing |
|                              | PRR-TTL/í | TTL\(x\) (i-classes load rate) | probabilistic routing |
|                              | PRR2-TTL/í | (same policy as PRR) | probalistic routing |
|                              | PRR2-TTL/í | two-tier domain partition | TTL\(x\) (i-classes load rate) |

Table 5.2: Summary of DNS dispatching algorithms.

Just exploring the dispatching component alone (constant TTL algorithms) is inadequate to address both node heterogeneity and non-uniform distributions of clients among domains.

- Adaptive TTL schemes can be easily integrated with even simple scheduling policies such as RR or RR2. This approach shows good performance for various node heterogeneity levels and system parameters, even in the presence of non-cooperative name servers.

- When there is full control on the choice of the TTL values, that is all (or most) name servers are cooperative, DRR2-TTL/S_K is the strategy to prefer.

- When there is limited control on the chosen TTL values, both DRR2-TTL/S_K and PRR2-TTL/K show reasonable resilient to the effect of non-cooperative name servers.

- Both DRR2-TTL/S_K and PRR2-TTL/K perform well, even if the domain load rate cannot be accurately estimated because of high variability in the load sources.

- The heterogeneity level of the Web-server system affects the achievable level of performance. Specifically, our results indicate that if the degree of heterogeneity is within 50\%, the probability of node overloading guaranteed by best adaptive TTL policies is always less than 0.05-0.10. Therefore, to achieve satisfactory performance, it would be desirable not to exceed this heterogeneity level in the
design of a distributed Web-server system.
Moreover, the adaptive TTL algorithms give the best results even for homogeneous Web-server systems. In these instances, they are almost always able to avoid overloading nodes even if the ADNS control on the client requests remains below 3-4% of the total load reaching the Web site.

5.7 Concluding Remarks

Although distributed Web-server systems may greatly improve performance and enhance fault-tolerance of popular Web sites, their success depends on load sharing algorithms that are able to automatically assign client requests to the most appropriate node. The main problems that DNS-based dispatching have to face are the limited DNS control on client requests which reach the Web-server system, and the non-uniform client distribution among Internet domains. The problems are further complicated when we consider the more likely scenario of a Web-server system consisting of heterogeneous nodes.

In this chapter, we first showed that extending DNS-based dispatching strategies adopted for the homogeneous node case [48] does not lead to satisfactory results. Therefore, we propose a different class of strategies, namely adaptive TTL schemes. They assign a different expiration time (TTL value) to each address mapping taking into account the capacity of the chosen node and/or the relative load weight of the domain which has originated the client request.

A key result of this chapter is that, in most situations, the simple combination of an alarm signal from overloaded nodes and adaptive TTL dramatically reduces load imbalance even when the Web server system is highly heterogeneous and the ADNS dispatcher controls a very limited portion of the incoming requests. Moreover, the proposed strategies demonstrate high robustness. Their performance is almost not affected even when the error in estimating the domain load is sizable (say 30%), and it is only slightly affected in the presence of some non-cooperative name servers.
Chapter 6

Load Sharing through Server-based Mechanisms

Web-server systems based on the DNS approach are gaining popularity in wide area load sharing thanks to the transparency and generality of the name resolution process. The main issue in DNS-based load sharing is with IP address caching at local and intermediate name servers, which limits the DNS control to a small fraction of the requests reaching the Web system. As a consequence, the ADNS has no means to intervene on load assignments to a Web server until the TTL expires. Therefore, bursts of requests arrive to the same server from a domain with a large number of clients during the TTL period, causing high load skews, which the ADNS cannot control [48, 57].

A first rough solution to the problem would consist in setting a zero or very low TTL value in the resource record provided by the authoritative DNS server. But, as discussed in Section 4.2.4, such a solution is non-feasible. The problem of limited control on the assignment of address requests is exacerbated by the high non-uniformity of the load from different client domains. So, DNS-based dispatching approaches require the study of sophisticated policies apt to avoid Web server overload, as those described in Chapter 5. However, these policies can only prevent new DNS assignments to an over-utilized server, which is exposed to load fluctuations originating by already made DNS assignments. Indeed, from the Web server point of view, bursts of requests can arrive from a domain to the same server during the TTL period, thereby causing high load imbalance.

This chapter focuses on a two-level architecture that integrates the coarse-grained DNS dispatching with a request redirection mechanism carried out by the Web servers
through HTTP redirection (see Section 4.2.3). We will propose a large set of alternative redirection schemes that aim at achieving load sharing in the two-level dispatching architecture and evaluate each of them through simulation experiments. We will defer the study of policies that minimize the impact of WAN network delays on user-perceived response times to the next chapter. In this chapter, we intend to focus on Web-server system performance and demonstrate how an architecture that integrates DNS dispatching with Web-server redirection mechanisms is able to distribute the load and handle high request rates, thus minimizing server overload.

In the proposed system architecture, the assignment decision occurs at two dispatching levels. The authoritative DNS server of the Web site executes the first-level assignment by acting on the address lookup that requires the translation of the site name into the IP address of one of the Web servers. The second-level (re)assignment is carried out by each Web server and acts on individual requests of Web pages, thus achieving a more fine-grained assignment control than the DNS-based one.

To keep the distributed nature of the Web-server system transparent to the users, the two-level architecture uses a single hostname within the site URL, while each server node has a separate IP address (this corresponds to the traditional mechanism for DNS-based dispatching). Furthermore, HTTP redirection is executed using the server IP address in the location field of the response header. If multiple hostnames were used in the redirection, not only the distributed nature of the Web-server system would be unveiled to users, but also a new DNS lookup would be required. The use of multiple IP addresses is not without drawbacks: it is not totally invisible to the user (as the browser typically displays the new URL after redirection, and not the old one that the user clicked on or typed in); it could also lead to URL-space contamination. On the other hand, it has a main advantage: avoiding an additional address lookup before the client connects with the new server, which can considerably impact on the user-perceived latency time, as pointed out in Section 2.2.

The algorithms herein proposed for a two-level architecture can apply, without any further extension, to a three-level architecture, where the Web servers are grouped into Web clusters and each cluster provides one VIP address visible to clients.

The rest of this chapter is organized as follows. Section 6.1 describes the design space for the redirection schemes. Section 6.2 presents various policies for a distributed Web-server architecture in which redirection decisions are periodical made by the ADNS. Section 6.3 analyzes an alternative solution where the redirection process is asynchronously activated by the Web servers. Section 6.4 focuses on the system model
and the distribution functions that can realistically describe the workload reaching a Web system. Section 6.5 presents the experimental results that compare various synchronous (periodic) and asynchronous (on-demand) activation mechanisms. Section 6.6 analyzes related work on redirection mechanisms for dynamic request assignment to distributed Web servers and compares them with our proposal. Section 6.7 summarizes the main results presented in this chapter.

### 6.1 Design Space for Redirection Schemes

In this section, we analyze the design space for the redirection algorithms following the terminology introduced in Section 4.3.4. Various alternative redirection schemes are possible: we will analyze those fully compatible with existing Web standards and protocols. In particular, the authoritative DNS server and Web servers of the system are the only entities that collect and exchange load information.

Table 6.1 shows a summary of the main factors of the design space for the redirection schemes analyzed in this chapter. The first group of factors is about activating the redirection process; it includes the activation trigger mechanism (synchronous vs. asynchronous), or when the redirection decision process is activated, and the activation decision process (centralized vs. distributed), or where the redirection decision process is activated. The second group is about status information used to implement the redirection scheme. This can be either the server load information and/or the domain load information. The third group is about carrying out the redirection policy. It includes the selection of the target server to receive the redirected requests, and the entities that are redirected, which can be either an entire domain, some individual clients within a domain or both.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Activation trigger</strong></td>
<td>Synchronous (periodic)</td>
</tr>
<tr>
<td>(when)</td>
<td>Asynchronous (on Web server demand)</td>
</tr>
<tr>
<td><strong>Activation decision</strong></td>
<td>Centralized (DNS)</td>
</tr>
<tr>
<td>(where)</td>
<td>Distributed (Web servers)</td>
</tr>
<tr>
<td><strong>Status information</strong></td>
<td>Server load (CPU queue and/or utilization)</td>
</tr>
<tr>
<td></td>
<td>Alarm (domain load (domain hit rate))</td>
</tr>
<tr>
<td><strong>Server selection</strong></td>
<td>Assignment Table</td>
</tr>
<tr>
<td>(how)</td>
<td>Assignment Table and Server Percentage List</td>
</tr>
<tr>
<td><strong>Redirected entities</strong></td>
<td>Domains (D)</td>
</tr>
<tr>
<td>(what)</td>
<td>Clients (C)</td>
</tr>
<tr>
<td></td>
<td>Domains and Clients (UC)</td>
</tr>
</tbody>
</table>

Table 6.1: Alternative design choices for redirection schemes.
Details on the various alternatives will be given in later sections. Let us now classify the different redirection approaches based on the activation trigger mechanism and the activation decision process. We assume that the activation decision is assigned to either the ADNS in the centralized decision case or any of the Web servers in the distributed case. The centralized decision is always combined with a synchronous (or periodic) activation, the so-called **centralized synchronous redirection** schemes (or, briefly, synchronous redirection). The distributed decision comes always together with an asynchronous or on-demand activation, the so-called **distributed asynchronous redirection** schemes (or, briefly, asynchronous redirection). Once a redirection decision has been made, the redirection process is always carried out by the Web servers.

Various algorithms with synchronous and asynchronous redirection schemes that take into account the other factors presented in Table 6.1 (i.e., status information, server selection for redirecting requests, and redirected entities) will be described in the next sections. All load sharing algorithms can be classified as **source-initiative** approaches [152].

Finally, we comment on the status information. In this chapter, the server load index can be either the utilization over a short interval, the CPU queue length or a combination of them. Instead, the domain load information is always measured as a **domain hit rate** (as discussed in Section 5.4), i.e., the number of hits per second reaching the Web system from a domain. Each Web server can estimate the hit rate from connected domains through the analysis of the logfile, that the server maintains to trace client accesses, and can periodically provide that information to the authoritative DNS server, as explained in Section 5.4.

All schemes proposed in this chapter are fully compatible with existing Web standards and do not require any code modification in the protocols, browsers, and domain gateways. Figure 6.1 shows the additional software components needed for the proposed DNS-based Web system with server redirection. In addition to the mapping base function, the authoritative DNS server includes the following components: a dispatcher, a load monitor, and a request collector. The dispatcher assigns each address request to one of the Web servers based on some dispatching algorithm as discussed in Sections 6.2 and 6.3. The load monitor tracks the load index from all servers, while the request collector gathers the domain load information from each server and estimates the domain hit rate of each connected domain.

Figure 6.1 also illustrates the corresponding set of components in one Web server
of the system. Besides the HTTP server, the proposed algorithms require a redirection component, a load checker, and a request counter, whose activity is related to DNS dispatching. The load checker tracks the server load index and sends this information to the local redirection component as well as to the ADNS load monitor component. The request counter estimates the number of requests received from each domain and periodically provides this information to the request collector in the authoritative DNS server. Finally, the redirection component determines if a client request has to be redirected and to which destination server, on the basis of information received by the ADNS dispatcher component and the load checker.

6.2 Synchronous Redirection Algorithms

This section describes various choices for the synchronous redirection algorithms. They have in common some features of the load information exchange (the information policy is a periodic one [141]), and are similar to the Global algorithm described by Zhou in [163].

The decision is centralized at the ADNS. Every \( t \) seconds each Web server sends some status information (server and/or domain load) to the ADNS. The ADNS gathers information from all the servers and builds the so called Assignment Table, where it specifies the Web server in charge of serving each connected domain. Therefore,
the interval \( t \) is referred to as the Assignment Table update interval. There are different approaches to build the Assignment Table. Some alternatives are described in Section 6.2.1.

The ADNS serves the address resolution requests by using the Assignment Table and broadcasts some load information and/or the results of its decision about redirection to all the Web servers. Since the Assignment Table determines the server responsible for serving a given domain, in order to resolve the address request the ADNS dispatcher has first to identify the domain from which the address request originated. This can be accomplished through the client clustering technique as described in Section 5.4.

The entities redirected by the synchronous policies can be either entire domains (SD), individual clients (SC) or even both (SDC). As the HTTP redirection mechanism works on an individual basis, domain redirection means that all clients belonging to the same domain are subject to the same redirection decision\(^1\). SD and SDC policies require that each server receives the Assignment Table from the ADNS dispatcher. For each page request, the Web server first identifies the domain to which the requesting client belongs; then, it checks the current Assignment Table to verify if it has to serve or to redirect the requests coming from that domain. In the latter case, the server redirects the page request to the target Web server designated in the Assignment Table.

Domain redirection requires that the Web server identifies which domain the requesting client belongs to. This can be accomplished by either applying the same client clustering technique deployed at the ADNS to the Web server or having the ADNS communicate the set of client addresses in each domain within the Assignment Table to the Web servers. The first approach requires an additional computation on the server, while the latter needs a more expensive communication among the authoritative DNS server and the Web servers. The rest of this section describes the various synchronous policies in which the assignment decision concerns different sets of clients requests, that is entire client domains and/or individual clients.

---

\(^1\)The name-to-IP address mapping in the client’s cache, which receives the redirection message, is automatically modified. Hence, all subsequent requests of the session from this client will go directly to the newly assigned server. On the other hand, the other clients of the same domain are not affected by this redirection message, because the IP address cache of the local name server has not been modified. Hence, when an entire domain is redirected, the Web server will need to redirect every client from that domain through the HTTP redirection mechanism.
6.2.1 Domain Redirection

First, we consider the case of the redirection entity being the entire domain. We assume a system in which every server periodically sends to the authoritative DNS server some status information, such as its own server load (e.g., the number of requests served in the last period) or/and the requests load from each domain to that server (i.e., the number of hit requests received in the last period). A detailed cost evaluation is out of the scope of this chapter. We can summarize that gathering server or domain load information at the ADNS causes an equal communication overhead. On the other hand, the domain load information causes a heavier computation overhead because it requires a periodic analysis of logfiles.

The first interesting issue to examine is the implication of different status information on the algorithms to build the Assignment Table. This analysis will not be repeated for the other redirection schemes since the same results apply.

**Domain Load Information Based Algorithms**

This class of algorithms uses just the domain load information to build the Assignment Table. Through the domain load information collected from each server, the ADNS estimates the domain hit rate for each connected domain, and ranks the domains from the most to the least popular on this basis. Various reassignment algorithms are feasible. We consider the following two schemes representative of different approaches to build the Assignment Table.

**RR-D.** The previously ordered domains are assigned to the servers in a round-robin way.

**Bin-D.** The potential load assigned to each server is characterized through a bin. It contains the sum of the hit rates of the domains assigned to that server by the Assignment Table under construction. The assignment policy aims at equalizing the bin levels through a greedy approach. In the first phase, one domain is assigned to each server, until all servers get a domain. Then, the residual domains are assigned by selecting the server with the lowest bin level every time. At each domain-to-server assignment, the hit rate of the assigned domain is added to the bin of the selected server.

**SD_Random.** Just for comparison purposes, we have an algorithm that provides a random assignment of the domains to the servers.
Server Load Information Based Algorithm

This class of algorithms builds the Assignment Table using just the server load index that each server periodically communicates to the ADNS. A simple scheme is the following. First, the ADNS ranks the servers from the least to the most loaded. Then, it assigns each connected (not ordered) domain to a server starting from the least loaded server in a cyclic fashion. This is referred to as **Bin-S** algorithm. We will see in the performance results section that the best server load index is that of the most recent server utilization evaluated over a short interval. Still, independently of this result, we can anticipate that algorithms using server load information only do not perform well at all: for this reason we will not consider them anymore. In conventional distributed systems, an up-to-date server load index usually provides a good indication of the future load condition [64, 96]. On the contrary, this information becomes obsolete rather quickly in a distributed Web-server system as it cannot reflect how many future requests will arrive due to the effects of address caching effects on past assignments and the high variability of domain connections.

Domain and Server Load Information Based Algorithm

This class of algorithms uses both domain and server load information to build the Assignment Table. The Bin-D algorithm in Section 6.2.1 can be generalized to consider both types of information. In this case, the algorithm is referred to as **Bin-DS**. The ADNS estimates the domain hit rates, and orders the domains from the most to the least popular on this basis. Through some server load index, the ADNS also orders the servers from the least to the most loaded. As third step, the ADNS builds the Assignment Table through the server bins and domain hit rate information. The server bin is updated according to the domain load after each assignment. In the first phase, the most popular domain is assigned to the least loaded server and so on until each server gets one domain. The other domains are assigned by selecting every time the server with the lowest bin level. The idea is that assigning the hottest domains to the least loaded servers would balance the load better than it could be achieved by Bin-D where the initial blind domain assignment could overload some servers.

6.2.2 Client Redirection

In the domain redirection algorithms, the ADNS decides about (re)assignments of entire domains to servers. Since the domain hit rates present high skews especially
for the hottest domains, it is almost impossible to get a perfect balancing. A more fine-grained redirection could be achieved by working on individual clients rather than of entire domains.

The class of client redirection algorithms builds the Assignment Table following the same method as the Bin-DS scheme described in Section 6.2.1. However, the table is now used only in the first-level assignment, which is carried out by the ADNS when it receives a request of address resolution.

The redirection (or second-level assignment), carried out by the Web servers, is based instead on the so-called Server Percentage List, which indicates the percentage of client requests that need to be redirected. This list is built by the ADNS as follows. First, the ADNS estimates the average bin level across all servers. Then, for the servers with a bin level lower than a certain range above the average (the value of this range being a parameter of the algorithm), the ADNS assigns a server percentage of 0% in the Server Percentage List. For the servers having a bin level exceeding the range above the average, the ADNS assigns the excess as their server percentage. Let us assume as an example that the range is equal to 20% of the bin level average and that the servers WS1, WS2, and WS3 have a bin level of 6, 13, and 11, respectively. The server WS2, having a bin level 30% higher than the average and exceeding the range, is assigned a server percentage equal to 30% in the Server Percentage List. The server WS3, having a bin level 10% higher than the average but not exceeding the range, is assigned a server percentage equal to 0%, as well as the server WS1.

Once obtained the Server Percentage List, the ADNS broadcasts it to each server. This list is used to implement a probabilistic redirection mechanism, based on individual clients, as follows. Web servers that have a server percentage equal to 0% do not reassign any request, while Web servers with a server percentage higher than 0% at each page request generate a random number \( p \) uniformly distributed between 0 and 1. If \( p \) comes higher than its server percentage (in the example WS2, if \( p > 0.3 \)), the server will return the required information; otherwise, it will redirect the request coming from that client to another server. We will consider three possibilities for the location policy, that is the policy to determine which server that has to receive the redirected request:

**SC RR.** Client requests are reassigned in a cyclic round-robin way to all the servers having percentage equal to 0% in the Server Percentage List.

**SC PRR.** Client redirection is done in a probabilistic round-robin (PRR) way,
where the probability is based on the available server capacity resulting from the latest server load information. This information can be easily broadcast by the ADNS together with the Server Percentage List.

**SC_LL.** The client requests are always reassigned to the server with the least load (LL); the ADNS would need to indicate the least loaded server when broadcasting the Server Percentage List. This third option is taken into consideration for comparison purposes only.

In comparison with domain redirection, it is to be noted that client redirection does not require that the Web server identifies the domain which the requesting client belongs to. Therefore, it avoids either an additional computation on the server or an extra communication among the authoritative DNS server and the Web servers.

### 6.2.3 Domain and Client Redirection

When the redirection of entire domains is too coarse an intervention on the load distribution, and if the redirection of individual clients does not work for the opposite reason, an effective alternative is a combination of the two methods. A mechanism to redirect domains and individual clients requires the ADNS periodically to broadcast both the Assignment Table and the Server Percentage List to the Web servers. As in SD algorithms, the Assignment Table is not only used by the ADNS for the first-level assignment, but also by the Web servers for the second-level one.

On the arrival of a page request, each Web server, after having identified the domain which the requesting client belongs to, checks the current Assignment Table to verify if it has to serve the requests coming from that domain. If not, the server redirects the requests according to the Assignment Table. Otherwise, it checks the Server Percentage List. In case its percentage is equal to 0%, the server serves the request and returns the required information back to the client. Otherwise, it implements one of the probabilistic redirection mechanisms presented in the previous section, that is **SDC_RR, SDC_PRR** or **SDC_LL**.

Table 6.2 summarizes the abbreviations of the synchronous redirection algorithms, of which we will evaluate the performance results in Section 6.5.

### 6.2.4 Alarm Messages

Any of the synchronous algorithms previously presented can be combined with a feedback alarm mechanism. When a server finds that its load is exceeding a threshold
Section 6.3. Asynchronous Redirection Algorithms

<table>
<thead>
<tr>
<th>Redirection entity</th>
<th>Server selection</th>
<th>Load index</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD (domain)</td>
<td>_Random</td>
<td></td>
</tr>
<tr>
<td></td>
<td>_RR</td>
<td>-D (domain load)</td>
</tr>
<tr>
<td></td>
<td>_Bin</td>
<td>-D (domain load)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-S (server load)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-DS (domain and server load)</td>
</tr>
<tr>
<td>SC (client)</td>
<td>_RR</td>
<td>-DS</td>
</tr>
<tr>
<td></td>
<td>_PRR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>_LL</td>
<td></td>
</tr>
<tr>
<td>SDC (domain and client)</td>
<td>_RR</td>
<td>-DS</td>
</tr>
<tr>
<td></td>
<td>_PRR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>_LL</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Synchronous redirection policies. (Server load information can be CPU queue length combined with utilization or not. For SC and SDC schemes the -DS suffix is not indicated explicitly.)

The feedback alarm mechanism outlined in Section 6.2.4 can be used to activate the redirection process itself. This leads to a new class of distributed reassignment schemes that are asynchronously activated on Web server demand.

No Assignment Table needs to be generated in this case. The Web system remains a typical DNS-based one where the authoritative DNS server carries out the first-level assignment through a RR or RR2 scheme. This DNS assignment process is integrated with a second-level (re)assignment mechanism triggered by any overloaded server. Threshold-based load balancing policies are quite popular in distributed computer systems [131, 139, 141, 163]. They have shown to be useful especially when jobs are independent and consist of single threads of control, which is a common feature for Web requests.

The ADNS, acting as a centralized collector, maintains the so called *Available Server List*, which is the list of servers that are not overloaded at that moment. This list is transmitted in reply to a server alarm message that the server sends to the ADNS when its utilization has exceeded a given load threshold. Thus, the components of
this class of distributed reassignment schemes can be classified in the following way: a *threshold-based* activation policy, and *demand-driven, sender-initiated* information policy [141].

Each overloaded server may redirect its page requests to any server of the Available Server List through the same HTTP redirection mechanism that is used in synchronous algorithms. A target server to receive the redirected request can be selected from the Available Server List in different ways. In this chapter, we consider only the stateless round-robin as location policy, having seen from preliminary results that the choice of a simple algorithm does not affect system performance significantly. Still, alternative schemes for the location policy will be discussed in the next chapter.

The redirection mechanism can be considered purely distributed because the ADNS has now taken the simple role of an information collector/communicator. This approach aims at reducing communication overheads determined by the redirection algorithm without changing the distributed nature of the algorithm itself. Indeed, the same effect could be obtained by excluding the ADNS and having each server broadcast the alarm message to the others, every time the server load threshold is exceeded.

The client redirection activated by heavily loaded servers can overcome the TTL constraint. In fact, with IP address caching at name servers, the DNS-based dispatcher loses direct control on the subsequent client requests arriving at the assigned server during the TTL period that follows the name-to-address mapping. So, it takes longer for the overloaded server to recover because the DNS policy can only stop the new DNS assignments to the overloaded server. There are no means to remove the already made assignments until the TTL expires. On the other hand, redirection can free the over-utilized server from a fraction of the previously assigned requests, before TTL expires.

### 6.4 Simulation Model

This section describes the simulation model that has been designed to evaluate the performance of the proposed algorithms, while experimental results from the simulator are presented in Section 6.5.

Since this chapter focuses on Web-server system performance, we will not model the Internet traffic [45, 51]. Instead, we will consider the major components that affect the performance of the Web-server system:
• the authoritative DNS server that dispatches some address requests through the name-to-address mapping;

• the local and intermediate name servers that through their IP address caching limit the control of the DNS dispatcher to few percentage units of all requests reaching the Web system;

• all the details concerning a client session, that consists of an IP address lookup phase followed by a Web page request phase.

The IP address request is initially submitted to the local name server of the client domain. If the cache of the local name server does not have a valid mapping for the site hostname, the mapping request is submitted to subsequent intermediate name servers, and only if the mapping is not cached in any of them, the request reaches the authoritative DNS server of the Web system, which returns the IP address of one of the servers and the TTL value. Each name server along the path from the ADNS to client’s domain caches this mapping for the TTL period.

During the Web page request phase various pages are requested to the Web system, initially to the selected Web server, and then to the same server or other servers if some redirection mechanism is activated.

The model incorporates all main characteristics of real Web workload by adhering to the measured statistical properties of typical client requests. High variability and self-similar nature of Web traffic are modeled through heavy-tailed distributions. A random variable $X$ has a heavy-tailed probability distribution if its upper tail declines as a power law, i.e., $P[X > x] \sim x^{-\alpha}$ as $x \to \infty$, where $0 < \alpha < 2$ [52]. Random variables following a heavy-tailed distribution exhibit very high variability; indeed, their variance is infinite, and, if $\alpha < 1$, their mean is also infinite. Typical heavy-tailed distributions proposed for modeling Web workloads are Pareto and Weibull distributions [13, 24, 51, 52, 130]. Heavy-tailed distributions appear to fit many recent measurements of computing systems, not only those related to Web traffic. For example, Unix process CPU requirements have been shown to follow an heavy-tailed distribution [74].

Table 6.3 summarizes the probability density function (PDF) $f(x)$, the cumulative distribution function (CDF) $F(x) = P[X \leq x]$, and the parameters range of some distributions we used in our workload model [99].

The number of page requests per client session, that is the number of consecutive Web pages a user will request to the Web-server system, is modeled according to the
inverse Gaussian distribution [77], where $\mu = 3.86$ and $\lambda = 9.46$.

The time between the retrieval of two successive Web pages from the same client, namely the user think time, is modeled through a Pareto distribution [13, 24], where $\alpha = 1.4$ and $k = 2$. Heavy-tailed distributions have been suggested as being a cause of self-similarity in network traffic [154]. In the context of the World Wide Web, Crovella and Bestavros explain the self-similarity of Web traffic with the superimpositions of heavy-tailed ON/OFF periods [51]. Indeed, each user’s behavior is modeled as a bursty ON/OFF process, where ON periods correspond to the transfer of Web files at a regular rate, while OFF periods correspond to silent intervals between transmissions.

The number of embedded objects per page, that is the number of objects making up a Web page inclusive of the base HTML page and its referred files, is also obtained from a Pareto distribution with parameters $\alpha = 1.33$ and $k = 2$ [24].

The inter-arrival time of object requests to the servers, that is the processing and displaying time spent by the browser parsing a document component and preparing the next request, is modeled by a heavy-tailed function distributed as a Weibull where $a = 0.382$ and $b = 0.146$ [23].

The function that models the object size distribution, that is the distribution of the file sizes requested to a Web server (note that this measure differs from the distribution of the file sizes stored in the server’s file system), is obtained from a hybrid distribution, where the body is modeled according to a lognormal distribution, while the tail is given by a heavy-tailed Pareto distribution [13, 22]. The parameter values for the lognormal distribution are $\sigma = 1.705$ and $\mu = 7.640$, while those for the Pareto distribution are $\alpha = 1.383$ and $k = 2924$ [22].

In the experiments, the system utilization is on average kept to 66% of the capacity of the entire Web system. This value is obtained as a ratio between the system (or offered) load, that is the average number of bytes per second requested to the Web-server system, and the system capacity, that is the sum of the capacities of all Web servers in the system measured in terms of the number of bytes per second.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>PDF</th>
<th>CDF</th>
<th>Range</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lognormal</td>
<td>$\frac{1}{\sigma \sqrt{2\pi x}} e^{-\frac{1}{2}(\ln x - \mu)^2} \frac{1}{\sigma x^2}$</td>
<td>no closed form</td>
<td>$x &gt; 0$</td>
<td>$\sigma &gt; 0, \mu \in (-\infty, \infty)$</td>
</tr>
<tr>
<td>Inverse Gaussian</td>
<td>$\frac{\lambda}{2\pi x^2} e^{-\frac{1}{2}(\ln x - \mu)^2} \frac{1}{\sigma x^2}$</td>
<td>no closed form</td>
<td>$x &gt; 0$</td>
<td>$\lambda, \mu &gt; 0$</td>
</tr>
<tr>
<td>Pareto</td>
<td>$\frac{\alpha k^\alpha}{x^{\alpha+1}} e^{-\frac{\alpha}{\mu} (x/k)^\alpha}$</td>
<td>$1 - (k/x)^\alpha$</td>
<td>$x \geq k$</td>
<td>$\alpha, k &gt; 0$</td>
</tr>
<tr>
<td>Weibull</td>
<td>$\frac{a x^{a-1} e^{-(x/b)^a}}{b^a}$</td>
<td>$1 - e^{-(x/b)^a}$</td>
<td>$x &gt; 0$</td>
<td>$a, b &gt; 0$</td>
</tr>
</tbody>
</table>

Table 6.3: Heavy-tailed distributions.
transferred. The offered load is generated by an average number of 2500 clients, which are assumed to be partitioned among the domains based on a Zipf’s distribution, that is a distribution where the probability of selecting the i-th domain is proportional to \(1/i^{(1-x)}\) [165]. (The uniform distribution is obtained by setting \(x = 1\), while the pure Zipf’s function has parameter \(x = 0\).) This is because if one ranks the popularity of domains by the frequency of their accesses to the Web-server system, the distribution on the number of clients in each domain is a function with a short head (corresponding to big providers, organizations and companies, possibly behind firewalls), and a very long tail. For example, a workload analysis on academic and commercial Web sites shows that on average 75% of the client requests come from not more than 10% of the domains [14]. More recent analysis also confirm a Zipf-like distribution of clients among domains [91]. In most of our experiments, the clients are assumed to be partitioned among 50 connected domains based on a pure Zipf’s distribution (highest skew). Other experiments based on Zipf’s distributions with different parameters also support the main results of this chapter (see Section 6.5.4). Similarly, considering a much larger number of connected domains to emulate a more realistic scenario does not affect the main results of this chapter. Indeed, these additional domains would just make the Zipf distribution tail longer, which is a very low source of offered load for the Web system.

DNS policies typically use a TTL value set to 300 seconds, and the Web system is assumed to consist of 7 servers with homogeneous capacity. Each server periodically calculates its utilization information during an observation period of 16 seconds. The sensitivity analysis carried out in Section 6.5.4 and 6.5.5 shows that the main conclusions of our experiments are not affected by the choice of some parameters such as TTL, client distribution, and number of Web servers; it also proves that the consideration of a less skewed workload model, such as the widely used exponential model, does not alter the relative ranking among the proposed redirection policies.

A summary of the model distributions and parameters, used in our simulation experiments, is shown in Table 6.4.

### 6.5 Experimental Results

In this section, we first discuss the performance metrics used to compare the different algorithms. Then, we discuss the performance results achieved by the redirection algorithms.
### Table 6.4: Parameters of the system model.

<table>
<thead>
<tr>
<th>Category</th>
<th>Parameter</th>
<th>Value (default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web system</td>
<td>Number of servers</td>
<td>3-9 (7)</td>
</tr>
<tr>
<td></td>
<td>Single server capacity</td>
<td>$3 \cdot 10^{-7}$ second per byte</td>
</tr>
<tr>
<td></td>
<td>Average system utilization</td>
<td>0.066$/$s</td>
</tr>
<tr>
<td>Domain</td>
<td>Connected</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Client distribution among domains</td>
<td>Zipf ($x = 0.0$)</td>
</tr>
<tr>
<td></td>
<td>Time To Live (TTL)</td>
<td>300 seconds</td>
</tr>
<tr>
<td>Client</td>
<td>Number</td>
<td>2500</td>
</tr>
<tr>
<td></td>
<td>Web page requests per session</td>
<td>Inverse Gaussian ($\mu = 3.86, \lambda = 9.46$)</td>
</tr>
<tr>
<td></td>
<td>User think time</td>
<td>Pareto ($\alpha = 1.4, k = 2$)</td>
</tr>
<tr>
<td></td>
<td>Embedded objects per Web page</td>
<td>Pareto ($\alpha = 1.33, k = 2$)</td>
</tr>
<tr>
<td></td>
<td>Inter-arrival time of objects</td>
<td>Weibull ($\alpha = 0.502, k = 0.46$)</td>
</tr>
<tr>
<td></td>
<td>Object size</td>
<td>Lognormal ($\mu = 7.640, \sigma = 1.705$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pareto ($\alpha = 1.383, k = 2924$)</td>
</tr>
</tbody>
</table>

#### 6.5.1 Performance Metrics

The main goal of this study is to investigate how redirection algorithms impact on load sharing with the aim to avoid that some Web server in the system becomes overloaded. The load balance of the overall system is only an indirect goal, so popular metrics, such as the standard deviation of server utilizations, are not deemed useful for our purposes. Therefore, to evaluate the performance of the various redirection schemes, we use as main metric the *system maximum utilization* that has been defined in Section 5.6.1. So, we can get an indication of how well a redirection scheme works by tracking the period of time the system maximum utilization is above or below a certain threshold.

For figures showing sensitivity to system parameters, we use the probability that no server of the Web system is overloaded as performance metric. Typical is the 95 percentile of the system maximum utilization (or not exceeding 95% maximum utilization).

Another interesting performance parameter is the percentage of redirected page requests.

The simulator, based on the Independent Replication Method [99], was implemented using the CSIM18 package [108]. Each value is the result of five or more simulation runs with different seeds, and every run covering six hours of the Web-server system activity. Confidence intervals were estimated on all simulation results, and the 95% confidence interval was estimated to be within 5% of the mean.

The goal of our experiments is to measure how effective the redirection schemes are in improving the limited control of the ADNS on the address resolution requests,
so as to maximize load sharing in a distributed Web-server system. We will present
first the performance of synchronous and asynchronous redirection policies. Then, we
will compare the best policies of each class using a wide set of system parameters.

6.5.2 Performance of Synchronous Redirection Algorithms

In synchronous redirection schemes, the ADNS has to collect some status information
to build the Assignment Table. At interval, every Assignment Table update, each Web
server communicates information on server and/or domain load to the ADNS, which
replies through a broadcast of the Assignment Table and/or the Server Percentage
List. The overhead for these communications is negligible compared to the network
traffic due to client requests and is quite similar in any policy of state information
gathering; therefore, we do not include it in our system model.

Domain Redirection Algorithms

The first set of experiments evaluates the performance of synchronous domain redi-
rection schemes, using an Assignment Table update interval set to 60 seconds. In
Figure 6.2, SD_Bin-S is the policy using the most recent server utilization as server
load index, while SD_RR-D is the algorithm based on the domain load information.
Their variants with alarm messages are not shown because they give analogous re-
results. This figure reveals the poor performance of any scheme that uses the domain
or the server load information alone to build the Assignment Table. The probability
that no server is overloaded or over 95% of system maximum utilization ranges from
approximately 0.5 for SD_RR-D to something more than 0.6 for SD_Bin-S. However,
these values resemble the performance of the SD_Random scheme that makes no use
of status information.

Using either status information alone leads to poor results, hence in Figure 6.3
we consider the domain redirection schemes that use a combination of domain and
server information. As we assume that the domain load is always the domain hit
rate, the focus of this figure is on the server load index used by the SD_Bin-DS
algorithm. Three variations on the load index are taken into consideration: SD_queue
based on the number of requests in the server queue, SD_util based on the server
CPU utilization, and SD_queue&util based on the sum of the number of requests
in the server queue and the number served in the most recent observation interval
(to estimate the utilization). Although the curves of SD_util and SD_queue&util
are quite close to each other, this and other not shown results (where different load indexes were used, such as instantaneous CPU queue length, exponentially averaged CPU queue length [64, 96]) highlight that the most useful server load index is the utilization evaluated in the most recent observation interval.

Experiments for the other synchronous algorithms (i.e., client redirection, domain and client redirection) bring to similar conclusions. Thus, we do not report details of the experiments which demonstrate that:

- the most useful server status information is its utilization;
- it is preferable to combine server and domain load information to build the Assignment Table.

Henceforth, we will use said load information for all synchronous redirection schemes.

A cross comparison of Figures 6.2 and 6.3 shows that building the Assignment Table through a combination of domain and server information improves the performance remarkably: SD_Bin-DS using server utilization has a probability of 0.92 that no Web server will get to exceed 95% utilization, while both the SD_Bin-D and SD_Bin-S schemes have a probability below 0.5.

**Domain and Client Redirection Algorithms**

We consider the issue of redirection granularity next. Although the SD_Bin-DS policies achieve acceptable performance, the load control granularity obtainable by
Section 6.5. Experimental Results

the domain redirection algorithms is too coarse to obtain satisfactory results. The risk
is to have the largest domain reassigned to a different server that becomes overloaded
in few seconds. Moreover, as we will see next, SD policies give acceptable results at
the price of a frequent update of the Assignment Table and of a high percentage of
redirected requests.

A more fine-grained redirection is achievable with client redirection algorithms
(SC), as well as with domain and client redirection algorithms (SDC); the former re-
assign individual clients, while the latter reassign both entire domains and individual
clients. Figure 6.4 shows the performance of SC and SDC schemes, where the Web
servers can use RR, probabilistic RR, and LL (least load) algorithms for the selection
of which server to redirect the client requests to. The various schemes compared in
pairs are SC vs. SDC, RR vs. PRR, RR vs. LL. The main result is that policies com-
bining domain and client redirection work better than schemes which redirect clients
alone. On the other hand, there is no appreciable difference between RR and PRR
schemes. The LL algorithm assigning the request to the least loaded server does not
perform well as it fails to spread the load among multiple servers. As PRR requires
additional information with no evident performance improvement, hereafter we will
use SDC_RR as the basic algorithm for domain and client redirection. Figure 6.5
summarizes the performance of the best three synchronous algorithms with domain
(SD), client (CD), and domain and client (SDC) redirection. The figure shows that
client redirection combined with domain redirection improves substantially the per-
formance of the domain or client only redirection scheme. The probability that no
server is overloaded is almost guaranteed (i.e., higher than 97-98%).

![Figure 6.4: SC and SDC schemes with RR, PRR and LL reassignments.](image1)

![Figure 6.5: Comparison of best performing synchronous schemes.](image2)
Once seen that the synchronous redirection policies are able to effectively share the load in a distributed Web-server system, the next step points to tuning these algorithms at best. In particular, we will address the issue on how to minimize the communication overheads and reduce request reassignments. Indeed, if the Assignment Table update is short, the centralized activation of the synchronous redirection algorithms can cause high computation and communication overheads in the process of gathering status information, building the Assignment Table and broadcasting it to the servers. Hence, an important goal is to lower the frequency in updating the Assignment Table. We found that a good updating interval for the SC scheme should have a value close to the TTL chosen by the ADNS. In the reported experiments, the interval is set to 300 seconds. As this interval is high enough to limit communication overheads, while the percentage of requests redirected by SC is very low (i.e., about 4-5%), we will focus our optimization analysis on SD and SDC schemes.

![Figure 6.6: Sensitivity of SD and SDC schemes to the frequency of the Assignment Table updating.](image1)

![Figure 6.7: Sensitivity of the SDC scheme to the frequency of the Assignment Table updating.](image2)

Figure 6.6 compares the sensitivity to the Assignment Table update interval, being the probability that no server in the system is overloaded (i.e., exceeding 95% utilization) the performance metric. We consider both SD and SDC schemes, as well as their variants with the feedback alarm at ADNS (as discussed in Section 6.2.4). This figure shows the performance of the best SD and SDC schemes. The alarm threshold is set to 0.85. The maximum number of overloaded servers that can be excluded from the Assignment Table is set to half the total number of Web servers (if more than three servers are overloaded, the ADNS ignores their feedback alarms). Contrary to previous results, now the alarm mechanism does not improve the performance of the
Section 6.5. Experimental Results

Redirection policies.

The performance of the SD policy, with or without alarm, becomes very poor as the update interval of the Assignment Table increases. Instead, the SDC schemes without alarm are insensitive to the update interval. This stability is very important, because larger update intervals not only diminish the computation/communication overhead but also the number of reassigned requests. So we can limit the percentage of users that perceive an increase in the response time without affecting the performance for the SDC scheme. Figure 6.7 shows that increasing the update interval from 60 to 300 seconds causes a substantial reduction of reassigned requests (from 0.34 to 0.12) with no performance degradation.

6.5.3 Performance of Asynchronous Redirection Algorithms

In asynchronous redirection schemes, the ADNS has to collect alarm messages from heavily loaded servers. The utilization threshold that triggers the alarm message is set to 0.75. (Similar results were observed for different alarm thresholds such as 0.7 and 0.8.) For the shown results, each server evaluated if its utilization had exceeded the alarm threshold at intervals of 8 seconds; their lapse is referred to as the check-load interval.

![Diagram](image)

Figure 6.8: Asynchronous redirection schemes with various DNS algorithms.

An asynchronous client redirection algorithm (AC) can be described by specifying the first-level assignment scheme carried out by the ADNS and the second-level (re)assignment algorithm executed by the Web servers. However, as we found in experiments not shown here that different server location policies (e.g., RR or PRR) do not impact much on the system performance, we differentiate the AC al-
algorithms only on the basis of their DNS assignment scheme. In particular, we consider the round-robin (AC_RR), the round-robin combined with alarm from heavily loaded servers (AC_RR_alarm), and the two-tier round-robin with alarm messages (AC_RR2_alarm) schemes. The last two exclude the overloaded servers even from the DNS assignment.

In Figure 6.8 we compare the performance of the AC schemes to that of RR and RR2 with alarm where the DNS first-level assignment is not supplemented by the second-level server (re)assignment. The improvement in favor of any AC algorithm is considerable. The client redirection can overcome the drawbacks caused by the IP address caching mechanism. Even stateless schemes, such as RR that performs very poorly under skewed workload on client distributions, perform, when combined with a client redirection mechanism (e.g., AC_RR), better than stateful schemes at the ADNS (e.g., RR2 with alarm feedback [48]).

Figures 6.9 and 6.10 show that the asynchronous redirection schemes are very sensitive to the length of the check-load interval. The performance of the AC policies deteriorates from over 0.9 to 0.6 as the interval increases, while the percentage of redirected requests remains constant. Therefore, it is reasonable to use short periods such as 8 or 16 seconds because the server load evaluation does not necessarily imply an activation of the redirection mechanism. Furthermore, the communication overhead is just a packet-size message sent from the Web servers to the ADNS. This is in contrast to shortening the Assignment Table update interval in the synchronous redirection schemes, because it causes much higher communication and computation overheads.
6.5.4 Sensitivity to System Parameters

In the previous sections we demonstrated that the synchronous (SDC) and asynchronous (AC) algorithms are able to control the load of a distributed Web-server system fairly well.

We now compare, for various system scenarios, the best algorithm of both centralized synchronous SDC and distributed asynchronous AC to a theoretical version of the scheme adopted by other distributed Web systems using redirection. Unlike AC that triggers the redirection mechanism on server demand, the other proposed systems are based on distributed and asynchronous algorithms activated on client demand. Hence, just for comparison purposes, we implemented the following simplified version of AC that tends to reassign requests to the least loaded server, namely LLS. The ADNS distributes the requests in a round-robin way. Each server is periodically informed about the other servers load. (Since this comparison does not consider communication overhead, we can assume that total load information exchange occurs even every 4 or 8 seconds.) At each page request, the server checks its utilization. If it has overcome a given threshold, the client request is redirected to the least loaded server. (A naive version of this scheme where the reassignment is not triggered by a threshold mechanism, but it is potentially carried out at each client request gave very poor results, hence it is not presented here even if it resembles more realistically the policies proposed in other Web systems.)

![Figure 6.11: Performance and percentage redirection of best redirection policies.](image1)

![Figure 6.12: Sensitivity to client distribution among domains.](image2)

Figure 6.11 shows the probability that no server is overloaded and the percentages of redirected requests for various policies. The synchronous SDC algorithm is the clear
winner, because it guarantees best load sharing with the lowest redirection percentage. Asynchronous policies achieve similar (but worse) results only for very short check-load intervals (8 seconds for AC) or unrealistic total load information exchange (4 seconds for LLS).

The performance comparison is now carried out as a function of some critical system parameters, such as client distribution among the domains (Figure 6.12), TTL value (Figure 6.13), and number of Web servers (Figure 6.14).

Figure 6.12 compares the performance of redirection policies in a system where the distribution of clients among the domains varies from the pure Zipf \( x = 0 \) to the uniform distribution \( x = 1 \). The SDC algorithm achieves better results than the asynchronous AC scheme, and both are better than the LLS algorithm. The robustness with respect to the client distribution is important because client scenarios tend to change frequently in the real Web environment. Performance rank does not change also when we vary the TTL value, during which the hostname to IP address mapping is cached, and the number of servers in the distributed Web system.

![Figure 6.13: Sensitivity to TTL value for IP address caching.](image1)

![Figure 6.14: Sensitivity to the number of Web-servers.](image2)

### 6.5.5 Sensitivity to Workload Model

As final step, we evaluate how the realistic workload model described in Section 6.4 may affect the conclusions of this chapter. To this purpose, we consider another workload model where all heavy-tailed distribution functions specified in Table 6.4 are replaced with exponential and Poisson distributions. It is to be noted that this kind of model, although not as realistic for a Web environment, is commonly used in
studies on distributed systems [152].

![Table 6.5: Heavy-tailed vs. exponential workload model.](image)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Exponential model</th>
<th>Heavy-tailed model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web page requests per session</td>
<td>Exp. (mean 12)</td>
<td>Inverse Gaussian ($\mu = 3.86, \lambda = 9.46$)</td>
</tr>
<tr>
<td>User think time</td>
<td>Exp. (mean 15)</td>
<td>Pareto ($\alpha = 1.5, k = 2$)</td>
</tr>
<tr>
<td>Embedded objects per Web page</td>
<td>Uniform in [5-15]</td>
<td>Pareto ($\alpha = 1.3, k = 2$)</td>
</tr>
<tr>
<td>Inter-arrival time of objects</td>
<td>Exp. (mean 15)</td>
<td>Weibull ($\alpha = 0.382, \beta = 0.146$)</td>
</tr>
<tr>
<td>Object size</td>
<td>Exp.</td>
<td>Pareto ($\alpha = 1.383, k = 2924$)</td>
</tr>
</tbody>
</table>

All experiments were replicated under the exponential-based workload model, whose parameters are summarized in Table 6.5 (all time values are in seconds). Only the figures that show the most significant results are shown here, while more details as well as the description of the exponential workload model can be seen in [37]. In particular, the synchronous schemes with domain, client, domain and client redirection are compared in Figure 6.15, while in Figure 6.16 the best synchronous algorithm versus two asynchronous policies. These figures correspond to Figure 6.5 and Figure 6.11, respectively.

![Figure 6.15: Comparison of best performing synchronous schemes (exponential workload model).](image)

![Figure 6.16: Performance and percentage redirection of best redirection policies (exponential workload model).](image)

Besides an expected improvement of all the policies under a less skewed workload model, two surprising results came out.

For an exponential workload model, the SC policy performs better than the SD policy. Figure 6.16 shows that in this case SC is quite comparable to SDC. Other not reported results on sensitivity demonstrate that the best synchronous algorithm is still the one combining domain and client redirection, but the difference between
SC and SDC policies is not so evident as it is for a heavy-tailed workload model. For a heavy-tailed workload model, SDC appears clearly to be the best algorithm. The only way AC can reach an analogous performance is by shortening the check-load interval to few seconds. This implies, however, higher communication overheads. On the other hand, Figure 6.16 shows that for an exponential-based model it is not so clear whether an asynchronous is preferable to a periodic redirection scheme. Indeed, while SDC keep achieving analogous results for any workload model (this robustness is a great quality in the continuously changing Web scenario), AC improves much its performance when the client distribution is less skewed, to the point that it slightly overcomes SDC in many instances.

6.6 A Qualitative Comparison to Related Work

Various request routing and dispatching methods have been explored for geographically distributed Web-server systems, as described in Chapter 4. Unlike the policies proposed in this chapter, both Cisco DistributedDirector (CiscoDD) [44] and Distributed Server Groups (DSG) [70] have a totally centralized approach, where requests are assigned by a dispatcher through HTTP redirection. In particular, the DSG dispatcher selects the least loaded server [70], while the CiscoDD dispatcher uses a network metric to estimate which server is in closest proximity to the client that has originated the request [44] (see Section 4.4.1). Actually, both DSG and CiscoDD use redirection as the only means to assign requests: there is no further dispatching level, as there is in our proposals.

Closer to our algorithms, which use a two-level dispatching (DNS plus redirection), are those deployed in SWEB [8] and Distributed Packet Rewriting (DPR) [18, 30] systems. In SWEB, client requests are initially assigned by the ADNS in the round-robin manner. Then, each server may reassign a received request to any other server of the system through HTTP redirection (see Section 4.4.1 for more details). DPR uses simpler decision criteria, such as a hash function applied to the client packet address or the least loaded server policy. Most important, DPR, unlike all other approaches, reroutes client requests through packet tunneling instead of using the HTTP redirection mechanism.

Table 6.6 summarizes the main features of these schemes. In particular, we focus on the most important five parameters, four of which have been presented in Table 6.1; the fifth is the redirection mechanism, either HTTP or packet rewriting. A comparison
of the contents of this table points out the difference between our proposals and those existing in literature, where basically the redirection is used either in a completely centralized way or in a distributed way. No other combinations were yet explored.

6.7 Concluding Remarks

Replicating information among multiple Web servers is necessary to support high request rates to popular Web sites. In this chapter, we have studied a two-level Web system architecture in which the DNS-based dispatching is integrated with some server redirection mechanism. We have compared various alternatives with synchronous or asynchronous activation, and centralized or distributed decision on redirection. Moreover, we have analyzed coarse- and fine-grained reassignments (i.e., reassignments of entire domains or individual client requests) as well as different types of status information (i.e., server and domain load information), and server location policies for redirecting requests.

The performance results we obtained demonstrate that the DNS dispatching algorithms integrated with some redirection mechanisms are effective to avoid overloading Web servers, even in a realistic and highly skewed workload model. We observed that even stateless schemes, such as RR, that perform very poorly under skewed request load, can achieve, when combined with a client redirection mechanism, better performance than stateful schemes without redirection (e.g., RR2 with alarm feedback). In particular, our experiments indicate the following:

- The most useful status information to decide about reassignment is a combination of domain hit rate and server load (mainly, utilization over the latest short
interval).

- The redirection of individual client connections is necessary to better share the load in a distributed Web-server system. However, there are significant differences between asynchronous and synchronous schemes. Individual client redirection is sufficient to achieve acceptable performance for asynchronous schemes, but it is not satisfactory for synchronous algorithms, unless combined with domain redirection.

- The centralized synchronous algorithm gives the best results for a wide set of system parameters. However, we found no appreciable difference of performance with distributed asynchronous approaches, unless the Web-server system is subject to a realistic heavy-tailed workload model. Moreover, although the communication overhead of synchronous algorithms is typically higher than that introduced by asynchronous policies, the latter need a higher frequency of intra-system communications to achieve as good results. However, the centralized synchronous algorithm requires that both the ADNS and the Web servers identify which domain the requesting client belongs to.
Chapter 7

Minimizing User Response Time

The explosive growth in size and usage of the Web is causing enormous strain on users, network service, and content providers. Geographically distributed Web systems are the most scalable architectures to handle highly accessed Web sites. Distributed Web systems cannot rely, however, on the Domain Name System alone as the ADNS assignment is a coarse-grained distribution of the load among the Web server nodes. Moreover, heavy load fluctuations of Web workload are amplified by the geographical context.

To address these issues, in this chapter we investigate a distributed architecture that integrates DNS-based dispatching among the Web clusters with a third-level (re)assignment which can be activated by any Web server node that experiments some critical load condition. The system architecture consists of various Web clusters placed in strategic Internet regions. Each Web cluster uses two or more back-end Web server machines\(^1\) that are housed together in a location of an Internet region. Figure 7.1 shows an example of distributed Web site consisting of four Web clusters, each of them with multiple Web server nodes (WS). Web clusters are typically interconnected through a high speed backbone to facilitate cooperation and information exchanges among the centers. Each Web cluster provides a single VIP address that corresponds to the address of the front-end Web switch. The Web switch receives the totality of HTTP requests reaching the Web cluster and routes them to the different server nodes in the cluster.

In our three-level system architecture, the cluster selection is centralized and based on DNS mechanism, while the possible server reassignment is totally distributed on

---

\(^1\)We consider Web sites consisting of homogeneous clusters, where each server node owns or can access a replicated copy of the Web site content.
Web servers and executed through HTTP redirection. The authoritative DNS server (ADNS) of the Web site executes the first-level coarse-grained request assignment. After the address lookup phase, the page request arrives at the Web cluster selected by the ADNS and the cluster Web switch executes the second-level routing among the cluster server nodes. Intra-cluster dispatching algorithms are out of the scope of this chapter. Specifically, we consider a one-way layer-4 Web switch that dispatches requests according to the WRR strategy (see Section 3.3.2).

Through the redirection mechanism, an over-utilized server can get immediately rid of a fraction of the requests previously assigned by both the ADNS and the cluster Web switch (as transient load peaks on a server may also be due to the intra-cluster dispatching). Since redirection acts on individual requests of Web pages, it can achieve a more fine-grained control than DNS-based selection alone.

While in previous chapters we have focused on sharing the load among Web server nodes being the reduction of response time only an indirect goal, in this chapter we also address the key issue of minimizing the impact of WAN network delays on response times perceived by users. In Chapter 6, we have demonstrated that a second-level dispatching carried out through HTTP redirection is an effective mechanism to achieve load sharing. Nevertheless, redirection should be used selectively because it
may increase latency time experienced by users, especially when considering servers distributed over all Internet. Indeed, if network overhead is higher than system overhead, request redirection increases response time experienced by users. Hence, our second goal is to investigate whether it is possible to limit request redirection without affecting much on the system ability to achieve load sharing. The basic idea is to select the most suitable subset of requests to be redirected, that is, we investigate the selection component of the redirection algorithm. The overall goal is to guarantee that the network overhead due to redirection has an impact on user latency time inferior to the system overhead due to an overloaded Web server.

The study is carried out through a simulation model of the Web system and network infrastructure. The system model details all characteristics of Web client/server interactions, while the network model is an approximate Internet vision that should provide a fair testbed to compare performance of different algorithms for geographic load sharing.

The remainder of this chapter is organized as follows. Section 7.1 describes various algorithms for DNS-based cluster selection. Section 7.2 presents various strategies that let the Web servers redirect client requests and limit the percentage of redirected requests. Section 7.3 describes the simulation model that has been developed to analyze the performance of the geographically distributed Web system. In Section 7.4 we present and discuss extensive simulation results. Finally, Section 7.5 summarizes the contributions of this chapter.

### 7.1 DNS-based Dispatching Algorithms

In this section, we present some algorithms that allow the authoritative DNS server to select an appropriate Web cluster. The policies considered can be grouped on the basis of the amount of system information being used, as summarized in Table 7.1.

**RR.** The ADNS uses no system information to decide about selection. The address requests are assigned in a round-robin manner among the Web clusters.

**Load.** Under this policy, the authoritative DNS server makes the selection decision based on some load information about the Web clusters. The address request is assigned to the cluster having the lowest load index. The *cluster load index* provides to the ADNS a global load information regarding the overall cluster. It can be either the mean value of the load indexes of all servers in the clus-
ter, or the maximum value among the server load indexes. As pointed out in
Section 3.3.2, different metrics can be used to estimate the server load index;
among the others, we select the total number of open TCP connections on the
server, and the utilization of the server node resources. In addition, the above
metrics can be combined using different weights and functions.

**Prox.** In a geographical context, the ADNS selection capability is typically extended
by taking into account the network location from which the client request origi-
nates. Using a proximity based algorithm, the ADNS is able to map with high
probability the site name into the VIP address of the Web cluster closest to the
client. DNS address caching augments this probability because intermediate
name servers of each Internet region will tend to get the resolution of the closest
cluster. Different proximity metrics can be used at the authoritative DNS
server to estimate which cluster is nearby the client. We will not address the
issue of evaluating network proximity; we refer to the discussion in Section 4.3.2.

**MinLP.** This policy represents the case of combining cluster load and network load
information. Clusters are ordered in two lists according to their load and net-
work proximity to the requesting client. Then, the ADNS selects the cluster
which minimizes the sum of the cluster position in the load and proximity list.
As an example, let us assume that clusters $C_1$, $C_2$, and $C_3$ are ordered by in-
creasing load as $C_3$, $C_2$, $C_1$, that is $load = [3, 2, 1]$, and by decreasing proximity
to the requesting client as $C_1$, $C_3$, $C_2$, that is $prox = [1, 3, 2]$. Then, the sum
of the cluster positions in the ordered lists is equal to $[4, 5, 3]$. Therefore, the
ADNS will route the client request to $C_3$.

<table>
<thead>
<tr>
<th>DNS algorithm</th>
<th>System information</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR</td>
<td>None</td>
</tr>
<tr>
<td>Load</td>
<td>Cluster load</td>
</tr>
<tr>
<td>Prox</td>
<td>Cluster-client proximity</td>
</tr>
<tr>
<td>MinLP</td>
<td>Cluster load and cluster-client proximity</td>
</tr>
</tbody>
</table>

Table 7.1: DNS-based selection algorithms.

To avoid problems arising from low TTL values, we assume that the authoritative
DNS server sets the TTL to a constant value equal to 300 seconds. We also consider
the error that may arise from inferring the client network location from the IP address
of the client’s local name server.
7.2 Server Redirection Algorithms

In this section, we present some algorithms that each Web server can use (independently of the implemented redirection mechanism) to select the new target cluster after that the redirection decision has been made. (Note that we do not consider intra-cluster redirection.)

Although in the previous chapter we have discussed redirection algorithms in which the activation of the redirection phase is assigned either to the centralized ADNS (i.e., synchronous algorithms) or to each Web server (i.e., asynchronous algorithms) and we have shown that both the methods achieve satisfactory load sharing, in this chapter we focus mainly on totally distributed mechanisms that do not require any centralized trigger or decision maker. Moreover, we also exploit further the location policies the Web server can apply to determine the new target cluster; unlike policies described in Section 6.3 in which the ADNS identifies the set of servers available for redirection, here we let each server identify independently the new target cluster.

The redirection process is activated on server-demand through an alarm mechanism that checks the current load of each server node. Any overloaded server starts the phase during which requests can be redirected when its load exceeds a given load threshold and ends the redirection phase when the load returns below the threshold. The server load can be measured using different load indexes, such as the number of active connections on the server, or the utilization of the server resources (e.g., CPU, disk, and application server). We use as load index the number of active connections, which is periodically evaluated every check-server-load interval.

Once the server has decided to activate the redirection process, the selection policy determines which requests have to be redirected. The reassignment is non-preemptive that is, only new requests for entire pages that have not yet been served are eligible for redirection. The straightforward solution is to redirect every new request reaching the overloaded server, as done by asynchronous algorithms described in Section 6.3 and by other redirection algorithms examined in literature so far [18, 100, 146]. However, this technique (for short, R-all) may cause more redirections than necessary to achieve good performance. Redirection increases the network impact on response time and incurs transfer overhead on the servers. To investigate how it is possible to limit the percentage of request redirection, we propose the following selection policies that redirect heaviest requests only. All the schemes are content-aware, that is they use some information contained in the HTTP request.
**R-size** redirects only requests for Web pages larger than a certain size. The motivation is that Web workload is characterized by a high variability [13, 22, 130]. Hence, a very small fraction of the largest files determines a large fraction of the load. We use the average size of a static Web page as the default size threshold for requests of static contents. In order to deploy this selection policy, each server maintains a table in which to each static Web page is associated the overall size of all the Web objects composing the page. However, this selection policy is feasible only for static content requests, as the impact of dynamic requests on the server load cannot be easily estimated.

**R-num** considers for redirection only those pages consisting of a large number of embedded objects (hits). We use the average number of hits in a Web page as the default threshold for deciding about redirection.

**R-dyn** selects for redirection only Web pages that require dynamic content generation. As dynamic content processing is more time- and resource-consuming than static content processing, the redirection of dynamic pages can shift away from the server a considerable amount of load.

Once decided that a request has to be redirected, the location policy selects the Web cluster that will receive the redirected requests. We consider four policies that are representative examples of diverse classes of approaches: the first policy is stateless, the second uses system load information, while both the third and the fourth use network load information.

**RR.** The first policy does not require information exchange among the Web clusters to make the decision. When a request is found eligible for redirection, it is sent to a Web cluster selected in a round-robin way.

**Load.** The second policy redirects the selected request to the Web cluster which has the lightest load, as observed in the past *check-cluster-load* interval. The deployment of this policy requires that the Web switches communicate periodically to each other the overall cluster load via the high speed backbone; then, each switch sends to servers in its cluster the global load information regarding the other clusters.

**CluProx.** The third policy redirects the selected request to the Web cluster that in the past *check-network-load* interval resulted best connected to the redirecting
cluster. It is to be noted that the deployment of this policy does not require that the Web clusters exchange messages each other; each Web cluster measures its own network proximity to the other clusters.

**Prox.** The fourth policy redirects the selected request to the Web cluster that in the past check-network-load interval resulted best connected to the requesting client. This policy requires that each cluster periodically evaluates its proximity to the clients from which it receives requests, and exchanges this information with the other Web switches via the high speed backbone.

Table 7.2 summarizes the server redirection policies we analyze in this chapter. We will denote the redirection algorithms by considering the selection and location policies. For example, **R-size_Load** is the algorithm that uses the size-based selection policy, and the load-based location policy.

<table>
<thead>
<tr>
<th>Name</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-all</td>
<td>None</td>
</tr>
<tr>
<td>R-size</td>
<td>Page size</td>
</tr>
<tr>
<td>R-num</td>
<td>Page hit number</td>
</tr>
<tr>
<td>R-dyn</td>
<td>Page content</td>
</tr>
<tr>
<td>Load</td>
<td>Cluster load</td>
</tr>
<tr>
<td>CluProx</td>
<td>Cluster-cluster network proximity</td>
</tr>
<tr>
<td>Prox</td>
<td>Cluster-client network proximity</td>
</tr>
</tbody>
</table>

Table 7.2: Server redirection algorithms.

### 7.3 Simulation Model

In this section, we describe the simulation model that has been developed to analyze the system performance. The model is designed to evaluate the integration of the proposed DNS- and server-based dispatching policies, whose objective is the reduction of the response time perceived by the users of a geographically distributed Web-server system.

#### 7.3.1 System and Workload Model

We divide the Internet into $K$ geographical regions located in different world areas. Each region contains a Web cluster, one or more authoritative DNS servers for the Web site (see Figure 7.1), and various client domains. The authoritative DNS server(s)
executes the first-level assignment by mapping the Web site name into the VIP address of one of the Web switches. The data requests arrive to the Web switch of the target cluster that executes the second-level assignment. At each page request, the Web switch applies a WRR dispatching algorithm. Each client page request is for a single HTML page that typically contains a number of embedded objects. Some objects may be dynamic that is, they require some computation and/or database search.

Each Web cluster consists of a front-end node that acts as content-blind Web switch, and two levels of server nodes. The nodes in the first tier work as Web servers, while the back-end servers in the second layer work as application servers and are used by the Web servers to satisfy dynamic requests. Each Web server has its own CPU, main memory, hard disk, and network interface. We use real parameters to setup the server resources. For example, the server disk is parameterized with the values of a real fast disk (IBM Deskstar34GXP) having transfer rate equal to 20 MBps, controller delay to 0.05 msec., seek time to 9 msec., and RPM to 7200. The Web server software is modeled as an Apache-like server, where an HTTP daemon waits for requests of client connections. Each application server is modeled as a black-box that provides dynamic content generation. Figure 7.2 shows the model of each server node.

![Figure 7.2: Model of the server node.](image)

We define the following time-dependent model to represent the variability of traffic coming from Internet regions, so that the most popular region can change during the simulation runs. Let $c_p_i(t)$ be the percentage of clients belonging to region $i$ at time $t$, where $\sum_{i=1}^{K} c_p_i(t) = 1$. This distribution function changes dynamically to take into account different day periods in the world. A given $c_p_i$ assignment is valid for the length of the period $t_p_e$. For each region $i$, the corresponding $c_p_i$ is regenerated at the beginning of each period, using the function reported in [12] and shown in Figure 7.3.
To take into account different world time zones, we assume that the time in region $(i + 1) \mod K$ is shifted of $t_K$ hours forward with respect to region $i$.

The simulator adopts an open system model in which client arrivals to the Web system follow an exponential distribution [153]. Each new client is first assigned to one Internet region with probability $q_i(t)$, and then is assigned to one client domain in that region through a Zipf distribution (with parameter $\alpha = 0.8$, corresponding to a highly skewed function [130]), which models the diverse domains popularity in each region.

The period during which a client visits the Web site is called session and consists of one or more page requests. At each request, the Web switch assigns the connection to an appropriate Web server. The workload model incorporates the most recent results on Web characterization. The high variability and self-similar nature of Web access load is modeled through heavy-tailed distributions such as Pareto, lognormal and Weibull distributions [13, 130, 153]. As explained in Section 6.4, this means that a random variable following a heavy-tailed distribution can take an extremely large value with non negligible probability.

The number of consecutive Web pages a user will request to the Web site (page requests per session) follows the inverse Gaussian distribution [11, 77]. The user think time between the retrieval of two successive Web pages is modeled through a Pareto distribution [11, 130]. The number of objects that make up a whole Web page, including the base HTML object and its in-line referred files, also follows a Pareto distribution [130].

Web file sizes typically show extremely high variability in size. The function that
models the distribution of the object size requested to the Web site varies according to
the object type. For HTML objects, it is obtained from a hybrid distribution, where
the body follows a lognormal distribution, while the tail is given by a heavy-tailed
Pareto distribution [12]. For in-line objects in a page, the size distribution is obtained
from the lognormal function [12]. Table 7.3 summarizes the probability mass function
(PMF), the range, and the parameter values we used in our workload model.

<table>
<thead>
<tr>
<th>Category</th>
<th>Distribution</th>
<th>PMF</th>
<th>Range</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pages per session</td>
<td>Inverse Gaussian</td>
<td>$\frac{1}{\sqrt{2\pi} \sigma} \exp \left( -\frac{(x-\mu)^2}{2\sigma^2} \right)$</td>
<td>$x &gt; 0$</td>
<td>$\mu = 3.86, \lambda = 9.46$</td>
</tr>
<tr>
<td>User think time</td>
<td>Pareto</td>
<td>$\alpha k^\beta x^{-\alpha - 1}$</td>
<td>$x \geq k$</td>
<td>$\alpha = 1.4, k = 1$</td>
</tr>
<tr>
<td>Objects per page</td>
<td>Pareto</td>
<td>$\alpha k^\beta x^{-\alpha - 1}$</td>
<td>$x \geq k$</td>
<td>$\alpha = 1.245, k = 2$</td>
</tr>
<tr>
<td>HTML object size</td>
<td>Lognormal</td>
<td>$\frac{1}{\sqrt{2\pi} \sigma} \exp \left( -\frac{(x-\mu)^2}{2\sigma^2} \right)$</td>
<td>$x &gt; 0$</td>
<td>$\mu = 7.630, \sigma = 1.001$</td>
</tr>
<tr>
<td>In-line object size</td>
<td>Lognormal</td>
<td>$\frac{1}{\sqrt{2\pi} \sigma} \exp \left( -\frac{(x-\mu)^2}{2\sigma^2} \right)$</td>
<td>$x &gt; 0$</td>
<td>$\mu = 8.215, \sigma = 1.46$</td>
</tr>
<tr>
<td>Dynamic object service time</td>
<td>Lognormal</td>
<td>$\frac{1}{\sqrt{2\pi} \sigma} \exp \left( -\frac{(x-\mu)^2}{2\sigma^2} \right)$</td>
<td>$x &gt; 0$</td>
<td>$\mu = 5.89, \sigma = 0.73$</td>
</tr>
</tbody>
</table>

Table 7.3: Workload model.

In this chapter, we consider two main classes of services provided by the Web site
that is, static services composed by requests for HTML pages with some embedded
static objects, and dynamic services, where some objects belonging to the page are
dynamically generated through Web server and application server interactions. We
model a dynamic Web page request as a page request with at most two embedded
objects that are generated by the application servers; the remaining embedded objects
are static.

A dynamic request includes the computation overheads on the application server
to generate the dynamic object and is characterized by a lognormal service time on
application server with mean equal to 0.5 seconds. The service time for a static object
is proportional to the file sizes shown in Table 7.3.

In this chapter, we consider two workload scenarios: in the static scenario static
services only are required to the Web site, while in the dynamic scenario the Web
site offers both static and dynamic services. In the dynamic scenario, the workload
composition consists of 80% of static page requests and 20% of dynamic page requests.

In the simulation experiments, we assume that there are $K = 4$ regions. Each
Web cluster is composed by 4 homogeneous Web server nodes (and 4 application
servers) for a total of 16 Web servers. Each simulation run lasts for 24 hours, and the
time difference among the regions is $t_K = 6$. (We also ran simulation experiments with diverse time differences among the Internet regions without observing substantial changes in results.) Figure 7.4 shows the overall traffic profile for one day under the dynamic scenario, where the batch size for counting the number of hits is set to 10 minutes and the client interarrival time is set to 0.025 seconds (that is, the client interarrival rate is 40 client/ sec).

The time to serve each client request includes all delays at the Web cluster, such as dispatching time, parsing time, service time for the page request and all embedded objects or the possible redirection time. If not otherwise specified, the check-server-load interval is set to 8 seconds, while check-cluster-load and check-network-load intervals are both set to 30 seconds. The threshold value of server load that triggers the redirection mechanism is measured in terms of active TCP connections and is set to 25.

7.3.2 Network Model

The network model aims at providing a controllable testbed where the transmission between Web clusters and clients has a cost. Our goal is to measure the impact of redirection on response time compared to request response time of not redirected requests. For this reason, we do not consider real Internet connections, network hierarchies, and narrow network bandwidth in the first mile. The model for communication delays is based on the following assumptions that, although simplified and subject to further improvements, introduce less arbitrary choices than pseudo-real network hierarchies and connections that could affect a fair comparison of the proposed algorithms.

In the model of client-server interaction, we refer to the HTTP/1.1 protocol that uses persistent connections and pipelining that is, the connection is left open between consecutive objects transmissions or at least for 15 seconds (after that timeout most servers typically close the connection) and the browser can issue multiple requests for embedded objects without waiting for a response [75, 121]. From [75], we have that the time to transmit $n$ objects belonging to the same page between region $i$ and $j$ is given by

$$T_{tr,n} = 2\text{rtt}_{ij} + \sum_{k=1}^{n}(S_{req_k}/ab_{ij} + S_{res_k}/ab_{ji})$$  \hspace{1cm} (7.1)$$

where $\text{rtt}_{ij}$ and $ab_{ij}$ are the \textit{round-trip time} and the \textit{available bandwidth} between region $i$ and $j$, respectively; $S_{req}$ and $S_{res}$ are the size of the client request and
server response for each object $k$, respectively. Equation 7.1 denotes that the time to transmit any message over the Internet is given by the time to establish a connection plus the ratio of the message size divided by the available bandwidth. For messages exchanged by Web clusters to communicate each other load and network information, the simplified form for Equation 7.1 is given by $T_{ir_1} = rtt_{ij} + S/ab_{ij}$, where $S$ is the message size.

Let us first discuss the message size in Equation 7.1 by distinguishing client request and server response. Although the traffic generated by Web clients is only about 6-8% of the global traffic (measured in bytes) that is, 1/9 of the traffic generated by Web servers [148], it is important to consider the client message component because of overheads of the HTTP redirection mechanism. Indeed, if a redirection occurs, then the client has to send two requests to two different servers in order to have its request fulfilled. The size of a client request $S_{req}$ follows a lognormal distribution with mean equal to 395 bytes ($\mu = 5.929, \sigma = 0.321$) [115] and typically fits a single packet. The server response consists of multiple files with various sizes following heavy-tailed distributions such as those described in Section 7.3.1.

The second parameter in Equation 7.1 is the available bandwidth $ab_{ij}$ that measures the communication delays between two Internet regions. We assume that these delays are due to a static factor (basic bandwidth) and a dynamic factor (traffic). The basic bandwidth $bb_{ij}$ between region $i$ and $j$ can be assumed deterministic. In particular, the basic bandwidth within a region and between this region and others is supposed to be large, medium-large, medium-narrow and narrow. The bandwidth values reported in Table 7.4 are taken from [75, 148].

<table>
<thead>
<tr>
<th>Bandwidth type</th>
<th>Basic bandwidth</th>
<th>Round-trip time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large</td>
<td>1.35 Mbps</td>
<td>[40, 70] msec</td>
</tr>
<tr>
<td>Medium-large</td>
<td>0.9 Mbps</td>
<td>[120, 150] msec</td>
</tr>
<tr>
<td>Medium-narrow</td>
<td>0.7 Mbps</td>
<td>[180, 210] msec</td>
</tr>
<tr>
<td>Narrow</td>
<td>0.4 Mbps</td>
<td>[270, 300] msec</td>
</tr>
</tbody>
</table>

Table 7.4: Network parameters.

We assume a good connection inside each region (that is, $bb_{ii} =$ large), and the same bandwidth in both directions (that is, $bb_{ij} = bb_{ji}$). To determine the bandwidth between the four regions, in the simulation experiments we consider the following values: $bb_{12} =$ large, $bb_{13} =$ medium-large, $bb_{23} = bb_{14} = bb_{24} =$ medium-narrow, $bb_{34} =$ narrow.
Section 7.3.  Simulation Model

We model the Internet traffic as a random parameter that reduces the basic bandwidth value. This random parameter changes dynamically to take into account world time zones and busy/quiet hours of each region. Let $\pi_{ij}(t)$ be a value between 0 and 1 that denotes how much fraction of the basic bandwidth can be used by a connection between region $i$ and $j$. Therefore, the available bandwidth at time $t$ is given by $ab_{ij}(t) = \pi_{ij}(t)bb_{ij}$.

Since the traffic depends on the number of clients in the regions traversed by a connection, we assume that the fraction of bandwidth available to a connection starting from a region and ending in another one, is related to the current popularity (i.e., to the number of clients) of the two end-point regions. To model this assumption, we define a discrete random variable $Z$, that represents the connection type and can take one of the following three values [118]: $lc$, that is lucky connection or light network traffic, $nc$, that is normal connection or mild network traffic, and $uc$, that is unlucky connection or heavy network traffic.

Let $\rho_i^Z(t)$ be the probability to have a connection of type $Z \in \{lc, nc, uc\}$ in region $i$, where $\rho_i^lc(t) + \rho_i^nc(t) + \rho_i^uc(t) = 1$. The distribution of $\rho_i^Z(t)$ is related to the popularity of region $i$; therefore, it is time-dependent. If at time $t$, region $i$ is highly populated, that is its originating traffic exceeds by 20% the uniform traffic distribution $1/K$, then the probability to have an unlucky connection traffic is considerable. On the other hand, if region $i$ has a low percentage of clients, that is its originating traffic is below 20% the uniform distribution, then the probability to have a light network traffic is high.

Let $\pi_i^Z(t)$ be the fraction of bandwidth for a connection of type $Z$ starting from region $i$. We assume that a connection can experiment a fraction of bandwidth varying in the interval $I_i^Z$, defined as $I_i^lc = [0.75, 1.0]$, $I_i^nc = [0.5, 0.75]$, $I_i^uc = [0.2, 0.5)$. Hence, $\pi_i^Z(t)$ is a random variable uniformly distributed in the interval $I_i^Z$, and is obtained in the following way. First, the type of connection $Z$ is determined by using the distribution for $\rho_i^Z(t)$, that is time and region dependent. Then, the value for $\pi_i^Z(t)$ is selected randomly according to the $Z$ value. Finally, the fraction of bandwidth for a connection between region $i$ and $j$ is given by $\pi_{ij}(t) = \frac{1}{2} \pi_i^Z(t) + \frac{1}{2} \pi_j^Z(t)$, where $i, j \in \{1, \ldots, K\}$. This choice takes into account different levels of popularity that any Internet region may have at time $t$. For example, if a lot of requests will arrive from region $i$ at time $t$, while region $j$ has a low number of clients, then the traffic for a connection between region $i$ and $j$ will result low/medium with high probability.
On the other hand, if both the regions are highly populated at time \( t \), then the connection is unlucky with high probability. Moreover, this network model includes the possibility that a client connected to the server through a large basic bandwidth may experiment unlucky connections due to heavy traffic. Dually, a client far from the server may experience a medium bandwidth thanks to a lucky connection.

For the same client session, we consider the same available bandwidth value because the route between two nodes is the same. Indeed, Paxson points out that Internet paths are heavily dominated by a single prevalent route and that about two-thirds of the Internet paths have routes persisting for either days or weeks [128].

The last parameter in Equation 7.1 to be discussed is the \textit{round-trip time}, that denotes the time necessary to establish the TCP connection between the client and the server inside and between Internet regions [75]. We set four intervals of round-trip delays, one for each type of bandwidth, and we choose randomly the round-trip time in the corresponding set. The interval values are reported in Table 7.4 and are taken from [4, 103].

Finally, we do not model the network delay due to the address resolution, that is the time needed to map the site name into the VIP address of an appropriate Web switch. This choice simplifies the network model, as we do not have to model explicitly the DNS infrastructure with all the intermediate name server nodes. This assumption is reasonable because the address lookup phase does not influence performance of investigated dispatching policies. However, we consider the address caching that occurs at name server as it influences the request assignment carried out at DNS level. In our model, the client initially submits the name resolution request to its local domain name server, and eventually to an intermediate or the authoritative DNS of the Web site. Furthermore, we also consider the address caching that occurs at the client side.

### 7.4 Experimental Results

In this section, we consider the objective of minimizing the response time experienced by users that access a geographically distributed Web site. Section 7.4.1 describes the metrics for comparing alternative dispatching policies, whose experimental results under the static and dynamic scenarios are discussed in Section 7.4.2 and Section 7.4.3, respectively.
7.4.1 Performance Metrics

The main objective of this study is to understand the impact on the user response time of distributing the Web service among the geographically distributed Web site with the aim to minimize it. Indeed, the crucial performance metric for users is the response time, as it directly correlates with the user's perception of the quality of service.

In highly variable systems subject to unpredictable traffic spikes such as Web systems, response time percentiles or even its cumulative distribution are more significant metrics than average values. Indeed, service level agreements (SLAs) in terms of performance are usually measured as the K-percentile of the server delay that must be less than Y seconds [93]. To evaluate which proposed dispatching algorithm guarantees quality of service, we use both the cumulative distribution of the page response time and its 90-percentile, that is the page response time limit that the Web site guarantees with 0.9 probability. The page response time corresponds to the interval elapsed between the submission of the client request for a given page and the arrival at the client of all objects corresponding to the page request. It includes TCP connection time, delays at Web server, network transmission time, and possible redirection overheads. The address resolution delay is not included in the page response time, since we do not model the address resolution time as stated in Section 7.3.2.

Since content provided by Web servers impose different server resources demands, we also differentiate between the response time of static and dynamic pages, where a dynamic page includes at most two objects dynamically generated as well as static objects containing images or other multimedia data. Another interesting performance parameter is the percentage of redirected requests because a further goal of this study is to propose and compare dispatching algorithms that minimize this value.

The simulator, based on the Independent Replication Method, was implemented using the CSIM18 package [108]. Each value is the result of ten or more simulation runs (each lasting 24 hours) with different seeds. The goal was that for all simulation results, the 95% confidence interval resulted to be within 5-6% of the mean.

7.4.2 Static Scenario

As most commercial products rely on proximity-based selection, our objective in the static scenario is to evaluate the performance of an enhanced ADNS that implements a proximity algorithm to reply to name resolution requests. Therefore, in this section
we assume that the ADNS selects always the closest cluster to the requesting client (being the client location estimated through the location of its local name server).

Figure 7.5 compares the cumulative distribution for the Prox algorithm (here indicated as DNS) with that of various redirection schemes based on the simplest selection policy (R-all) that lets an overloaded Web server to redirect all requests to a different Web cluster. This figure shows that even this naive server redirection approach achieves substantial performance improvement when compared to systems where dispatching is done by ADNS and cluster Web switches only. All redirection policies guarantee that the maximum response time is below 20 seconds while analogous performance is guaranteed to only 80% of requests assigned by proximity-based DNS algorithm. The 90-percentile of the best redirection policy is equal to 7 seconds that is, about the value that many studies and surveys consider the service level acceptable to most Web users [31]. The analogous service level is guaranteed to less than 70% of clients of Web systems based on two-level dispatching mechanisms (see DNS curve). Indeed, different time zones and non-uniform popularity of client domains cause some Web cluster to be swamped with requests originating from its region. The proximity-based DNS algorithm fails to distribute the load among more distant Web clusters that may be less loaded in that day period [133].

![Cumulative distribution of response time for the R-all selection policy.](image)

Figure 7.5: Cumulative distribution of response time for the R-all selection policy.

If we compare the three location policies (RR, Load, Prox), we observe that the circular reassignment among all Web clusters achieves best performance, the policy that reassigns requests to the least loaded cluster performs slightly worse, while the Prox location policy redirecting all requests to the closest Web cluster shows significantly worse results. The motivation for this results is that when we reassign all
requests, it seems preferable to spread the load among multiple Web clusters than concentrating all redirected load to one cluster, even if the receiver cluster is the lowest loaded or the closest. We observed that typically a burst of redirected requests improves performance on the sender cluster, but causes a temporary overload on the receiver cluster that, on its turn, activates the redirection mechanism. The consequence is that the Web system remains unstable for longer periods with tangible consequences on response time.

We next consider selection policies that limit redirection on the basis of the estimated size of the requested page (R-size) or number of hits (R-num). We plot the results of the former policy only, even because similar results are obtained for the latter algorithm. Figure 7.6 shows that redirecting only client requests having a size larger than the average page size improves for small response time values the performance achieved by the naive selection scheme. The relative performance order between Load and RR location policies is now inverted, and R-size_Prox performs much better than corresponding R-all_Prox. This confirms that redirecting only a subset of requests reduces system instability because the receiver cluster is not overwhelmed by bursts of additional requests. Hence, it becomes convenient to consider for redirection the least loaded cluster instead of proximity or circular assignments.

![Cumulative distribution of response time for the R-size selection policy.](image)

In the following experiments, we compare the performance of various selection and location policies by taking into account the 90-percentile of the page response time that is considered the most important parameter to evaluate quality of service of a Web system [93]. In Figure 7.7 the results are grouped according to the selection policy; different location policies are considered inside each group. We observe that
the Prox location policy remains the worst solution for all selection policies. RR location is fine to spread evenly the load when all requests are reassigned by overloaded servers, but this effect is less important when only a subset of requests is reassigned. When redirection is activated only on the most resource consuming requests, it seems useful to use some state aware location policy to select the most appropriate Web cluster. The Load location policy performs better than the stateless RR for both R-num and R-size selection schemes. Finally, it is important to observe that the 90-percentile of page response time for Web systems based on two-level dispatching (that is, DNS proximity and Web cluster dispatching) is equal to 32 seconds. This is more than six times the value achieved by the best redirection policies.

Performance shown in Figures 7.5, 7.6 and 7.7 are useful for algorithm comparison and must not be considered as absolute time values. The proposed network model is incomparable to the complexity of real Internet. In particular, we believe that redirection may have an impact on performance even greater than that shown in previous figures when a real geographical environment is considered. Consequently, the reduction of redirections achieved by R-num and R-size selection policies may limit network impact on latency time even more than that shown in Figure 7.7. To this purpose, in Figure 7.8 we show the redirection percentages for the selection and location policies considered in Figure 7.7. We can see that the naive selection policy tends to redirect close to 20% of the requests reaching the Web site, while the analogous metrics is below 5% for R-num and R-size selection policies. This difference is consistent if we consider overheads that each redirection causes on Web servers (e.g., time lost to open and close a TCP connection) and network (e.g., round-trip and new TCP connection delays).

To verify the robustness of the results, in the last set of experiments for the static scenario we analyze the sensitivity of the selection and location policies to some system parameters. In Figure 7.9 we analyze the 90-percentile of the response time as a function of the check-server-load interval for the best selection and location policies. As expected, performance of all policies tends to improve for lower values of this interval. It is reasonable to use relatively short periods such as 8 seconds, because server load evaluation does not necessarily imply the trigger of the redirection mechanism.

Figure 7.10 shows the 90-percentile of the page response time when the long-term utilization of the Web system varies from 0.5 to 0.75. All the policies that redirect requests show a similar behavior when the load increases. Although the limit for
Section 7.4. Experimental Results

Figure 7.7: 90-percentile of response time for various selection and location policies.

Figure 7.8: Redirection percentage for various selection and location policies.

Figure 7.9: Sensitivity to the frequency of the check-server-load interval.

Figure 7.10: Sensitivity to the long-term system utilization.

The guaranteed response time doubles, in the Web environment subject to heavy-tailed arrivals, a 0.75 sustained utilization corresponds to a highly loaded system that would require more resources. The need of the third-level redirection mechanism is well motivated by the results of a Web site using DNS proximity algorithm only: the 90-percentile of the page response time for 0.5, 0.6, and 0.7 system utilization is equal to 10, 32 and 56 seconds, respectively. This shows that DNS only is not able to guarantee request response time even when the system utilization is subject to little increases.

We have also carried out some sensitivity analysis as a function of the network bandwidth capacity. Figure 7.11 shows the variation of 90-percentile of response time when the network capacity ranges from 25% to 200% of the original basic values.
shown in Table 7.4. By halving or doubling the basic bandwidth among all Internet
regions, we observe that this parameter does not affect our previous conclusions. The
components of the response time due to the server delay and the network transmission
(Service and Network, respectively) are shown in Figure 7.12.

![Graph showing response time vs bandwidth percentage]

**Figure 7.11:** Sensitivity to the network bandwidth of response time.

**Figure 7.12:** Sensitivity to network bandwidth of service and network components.

In summary, under the static scenario:

- When the authoritative DNS selects the cluster closest to the requesting client
  (that is, Prox policy), clusters are unbalanced and the response time increases
  significantly, because arrivals from each Internet region are highly variable
depending on population, time zones, and day hour.

- To achieve a scalable and balanced Web system, it is necessary to integrate DNS
  proximity and Web switch dispatching with the HTTP redirection re-routing
  mechanism that any Web server can activate at its need.

- The naive selection policy (RR_all) achieves substantial performance improve-
  ment when compared to systems where dispatching is done by DNS and cluster
  Web switches only.

- Selection policies that redirect only the heaviest page requests allow to reduce
  system instability because the receiver cluster is not overwhelmed by bursts of
  additional requests.

- The response time achieved by sophisticated selection policies does not signifi-
  cantly differ from that obtained by the naive policy. On the other hand, R-num
  or...
and R-size selection policies are able to considerably reduce the percentage of redirected requests, thus limiting the network impact on response time in a real environment.

### 7.4.3 Dynamic Scenario

In this section, we compare the performance of the DNS dispatching algorithms and server redirection policies under the dynamic scenario. While in the simulation results discussed earlier we focus only on a DNS-based proximity selection, in the dynamic scenario our objective is to evaluate the impact on page response time of the integration of DNS dispatching algorithms with server redirection policies. As the proposed strategies use diverse system state information at different dispatching levels, we aim to identify the best performing combination of system information when the Web system is subject to an increasing offered load. We will denote the combination of DNS dispatching and server redirection by considering the ADNS dispatching algorithm and the selection and location policies. For example, **Prox_R-all_Load** refers to DNS dispatching carried out through the Prox algorithm and server redirection ruled by the naive selection policy, and the load-based location policy.

In the first set of experiments, we focus on the first-level DNS dispatching (without any redirection mechanism) to confirm the need of introducing a third-level assignment carried out by the Web servers, which has been already highlighted by the experiments executed under the static scenario.

![Figure 7.13: 90-percentile of response time achieved by DNS and Web switch dispatching for all pages.](image1)

![Figure 7.14: 90-percentile of response time achieved by DNS and Web switch dispatching for static and dynamic pages.](image2)

Figures 7.13 confirms that the pure DNS assignment based on proximity (Prox)
between clients and Web clusters fails to distribute the load among more distant Web clusters that may be less loaded in that day period. Routing requests always to the closest cluster creates problems as the Web cluster located in the most popular region during each period of the day is swamped with requests originating from its region. Different time zones and uneven popularity of client domains cause the Web system to be highly unbalanced under a proximity-based dispatching policy. When a stateless DNS scheme such as RR is used, the load is distributed among multiple Web clusters rather than concentrated to one cluster, even if both the network proximity and cluster load factors are neglected. The Load policy that assigns all address requests to the cluster resulted to be the least loaded one in the last interval, allows to spread the load more evenly than the Prox policy, because the load information changes more frequently than the proximity information (that is, the proximity information is quite static). Combining load and proximity information in DNS dispatching improves the results achieved by using only proximity information; however, MinLP performs worse than RR and Load policies because the closest cluster impacts on the selection decision.

Figure 7.14 shows the response time achieved by the first-level assignment when static and dynamic pages are differentiated. We observe that the response time obtained by the Prox dispatching algorithm increases sharply both for static and dynamic Web pages. The proximity-based algorithm can be applied with success only when the system is subject to a low offered load (i.e., less than 30 clients per seconds). When the system load increases, the reduction in the network transmission cannot make up for the service delay on an overloaded server.

In the next set of figures, we evaluate the impact of integrating the DNS assignment with server redirection. We consider the naive selection policy (R-all) that lets an overloaded Web server redirect all requests to a different Web cluster and observe the response time behavior with different location policies. Figures 7.15, 7.17, and 7.19 confirm that even a naive server redirection approach, such as R-all_RR, achieves substantial performance improvement when compared to DNS based dispatching only approach.

If we compare the response time achieved by the different location policies (i.e., RR, Load, ChuProx, and Prox) under the various DNS assignment schemes, we observe the different impact of combining DNS assignment and location policy.

Under the DNS-RR scheme (Figures 7.15 and 7.16), the circular location policy among all Web clusters achieves a good performance, and the location policy that
reassigns requests to the least loaded cluster performs even better under high load (more than 45 client/sec). On the other hand, the CluProx location policy which reassigns requests to the Web cluster closest to the redirecting cluster shows significantly worse results even than the pure DNS assignment, as it fails to spread the redirected load. The motivation is that in the system there are two clusters (say, A and B) that are better connected to the other ones, and are well interconnected among them. (This situation resembles the real network environment, in which the East and West North America coasts are well interconnected to each other through high-speed Internet backbones, and have a good connectivity to West Europe.) As a consequence, clusters A and B receive the requests assigned by the ADNS selection (distributed in a round-robin manner among the clusters) plus a quota of exceeding requests reassigned to them by the other clusters. Since the proximity information does not change significantly, when the peak traffic hours regard regions A and B because of the time-of-day (see Figure 7.3), the two clusters located in those regions are overwhelmed by requests and start redirecting requests in their turns. As clusters A and B are well interconnected to each other, they are not able to get rid of the load as they redirect requests each other.

Figure 7.15: 90-percentile of response time achieved by DNS-RR and R-all selection policy for all pages.

Figure 7.16: 90-percentile of response time achieved by DNS-RR and R-all selection policy for static and dynamic pages.

As regards the Prox location policy, it seems to be the best performing one when we consider in Figure 7.15 both static and dynamic services. However, if we distinguish the response time for static and dynamic pages as in Figure 7.16, we observe that the result is not confirmed under very heavy load, as the response time for dynamic pages augments sharply. However, when the client arrival rate exceeds 48 client/sec
all the cluster resources are heavily utilized, in particular the application servers.

We next examine in Figures 7.17 and 7.18 the effect on page response time of assigning address requests to the least loaded cluster (DNS-Load policy). We observe that the circular reassignment among all Web clusters achieves the best performance for all pages as well as for static and dynamic pages. When the Web servers reassign all request, it is preferable to spread the load among multiple Web clusters rather than concentrating all redirected load to one cluster as done by the CluProx location policy. Indeed, the DNS-Load scheme already assigns a burst of requests to the lowest load cluster. If this cluster becomes overloaded, its servers start to redirect requests, which should be evenly spread among the other clusters. Under the CluProx policy, all exceeding requests on the least loaded cluster are redirected to the closest cluster only.

![Figure 7.17: 90-percentile of response time achieved by DNS-Load and R-all selection policy for all pages.](image1)

![Figure 7.18: 90-percentile of response time achieved by DNS-Load and R-all selection policy for static and dynamic pages.](image2)

Figure 7.19 shows the considerable results achieved by the redirection mechanism on the proximity-based DNS scheme, where the clusters are overwhelmed by “local” requests. The Load and RR location policies achieve comparable results, although RR shows lower sensitivity (especially for static pages) when the Web system is highly loaded.

In the next set of experiments, we consider the impact of the server selection policy on the page response time. We neither applied the R-size selection policy nor the R-hit selection policy to a dynamic scenario, as they are effective to redirect static page requests only. When the server is overloaded, the redirection of static pages does not shift away the load imposed by more resource-consuming dynamic requests; therefore,
Figure 7.19: 90-percentile of response time achieved by DNS-Prox and R-all selection policy for all pages.

Figure 7.20: 90-percentile of response time achieved by DNS-Prox and R-all selection policy for static and dynamic pages.

the action of both the R-size and R-hit selection policies is inherently limited. For this reason, we show only the results achieved by the selection policy based on the service type of the requested page, that is R-dyn.

Figure 7.21: 90-percentile of response time achieved by DNS- RR and R-dyn selection policy for all pages.

Figure 7.22: 90-percentile of response time achieved by DNS-RR and R-dyn selection policy for static and dynamic pages.

Figures 7.21 and 7.22 show that redirecting only page requests which entail dynamic content generation (the most resources consuming service in our system) improves the performance achieved by the naive selection scheme, besides reducing the redirection percentage. We also evaluate the impact on performance of the server load index used to activate the redirection. When redirection is limited only to dynamic requests, it seems more appropriate to select as server load index the utilization of the
application server (RR_R-dyn_RRwas policy) rather than the number of active connections (RR_R-dyn_RR policy). In not shown experiments, we also observed that the RR location policy is better suited to reassign dynamic requests rather than the Load location policy. Similar results were achieved under the other DNS dispatching policies.

We have also evaluated the impact of a low bandwidth network on the dynamic scenario. In particular, we reduced the network capacity (both static bandwidth and round-trip time values) to 25% of the previous case. It is worth noting that the component of the response time due to the network transmission is comparable for both static and dynamic pages.

If we distinguish the network and server delay component in the overall page response time under normal system load, we observe that when the network capacity is set to 25% of the original value the network component accounts for 75% of the overall 90-percentile of response time, while in the previous scenario it accounts for 40%. Figures 7.23 and 7.24 show the response time for the proximity-based DNS dispatching algorithm and R-All selection policy. For all DNS dispatching policies, we observed that the curves are translated upward (due to the higher network component) but there is no substantial change from the higher bandwidth scenario.

![Figure 7.23](image1.png) **Figure 7.23:** 90-percentile of response time achieved by DNS-Prox and R-all selection policy for all pages (*low bandwidth network*).

![Figure 7.24](image2.png) **Figure 7.24:** 90-percentile of response time achieved by DNS-Prox and R-all selection policy for static and dynamic pages (*low bandwidth network*).

In summary, under the dynamic scenario:

- Due to the coarse-grain dispatching and highly variable system conditions, DNS
dispatching alone is not able to achieve a 90-percentile of the response time lower than eight seconds, which is considered to be the service level acceptable to most Web users [31]. Nevertheless, RR and Load algorithms achieve a better performance than a proximity-based policy, as the latter is not able to spread among the Web clusters the requests originated from the peak load region at a given time-of-day.

- To achieve a scalable Web system and guarantee an acceptable service level, it is necessary to integrate DNS and Web switch dispatching with a third-level dispatching activated by any Web server at its need.

- Selection policies that redirect only dynamic page requests allow to reduce user-perceived response time as well as system instability especially under heavy load conditions as they shift away from the first assigned cluster a considerable resource-consuming load.

- A lower network capacity does not alter the main results observed under the default network parameter values, which are reported in Table 7.4.

7.5 Concluding Remarks

Highly accessed Web sites require a distribution of content and servers over geographical regions to avoid network bottlenecks. Various proposals consider a geographically distributed architecture where the ADNS of the Web site evaluate network proximity and requests reach the closest Web cluster, where a Web switch executes the second-level dispatching.

In this chapter, we demonstrate that serving closest requests only may cause unbalanced clusters and may increase considerably system impact on response time, because arrivals from each Internet region are highly variable, depending on population, time zones, and time-of-day. To achieve a scalable and balanced Web system and provide to end users a satisfactory performance, we integrate DNS proximity and Web switch dispatching with an HTTP redirection mechanism that any Web server can activate at its need. We show that this third-level dispatching, when integrated with some limitation on request redirection, is a powerful mechanism to enhance quality of service of geographically distributed Web sites.
Chapter 8

Conclusions

In this thesis we address the problem of handling heavy traffic loads through the design of Web systems based on multiple server nodes as they constitute a viable and cost-effective approach to provide performance, scalability, and availability in Web services. We first focused on solutions where the request assignment decision occurs only at the Domain Name System level during the address lookup phase and then explored multi-level Web systems architectures, where the DNS routing is integrated with a redirection mechanism. The latter, introducing a further level of routing and dispatching along the course of the Web transaction, allows for fine-grained control on request assignment. In this chapter we summarize our work and discuss conclusions and future work.

8.1 Summary of Contributions

This section summarizes the contributions made by this thesis. These are:

- The proposal of DNS-based dispatching algorithms that address the issue of system load sharing. The proposed algorithms exploit the use of both parameters controlled by the authoritative DNS server of the Web site, that is the IP address of the selected server node and the period during which this address mapping is valid. Previous work on DNS-based selection operated on the server address only, at most setting the TTL parameter to very small values in order to increase the limited control on request assignment. This gross reduction of the validity of cached address mappings is, however, not feasible as it may shift the performance bottleneck from the Web system to the DNS infrastructure. We demonstrated through extensive simulation experiments that DNS algo-
rithms that tailor the TTL value adaptively for each address request by taking both client and server state information into account are able to avoid nodes overloading effectively even when the server nodes are highly heterogeneous. As the proposed DNS dispatching policies require an estimate of client state information with low computation and communication overhead, we also analyzed the feasible sources and types of information needed to a DNS-based architecture working in a realistic environment.

- The proposal of DNS-based dispatching algorithms integrated with server-based redirection algorithms in two-level Web system architectures. As DNS-based dispatching is intrinsically limited being the address resolution performed only once at the beginning of the Web session, we investigated server redirection algorithms to avoid load fluctuations originating by the first-level assignment. We proposed and evaluated a two-level Web system architecture that is able to ensure system load sharing. We analyzed a wide variety of alternatives, that is synchronous and asynchronous activation of redirection, centralized and distributed decision on redirection, coarse- and fine-grained request reassignments, different types of system state information. We showed that our solutions are effective to overcome sudden surges of load on server nodes thus avoiding overload periods, even under a realistic and highly skewed workload. These results suggest that redirection mechanisms are a useful method to augment considerably DNS-based routing and dispatching capability.

- The proposal of server-based redirection algorithms within a three-level Web system architectures. Based on our results about redirection algorithms in a two-level architecture and to exploit the performance advantages of geographical distribution, we proposed and evaluated a three-level system architecture and the integration of routing mechanisms and dispatching algorithms aiming at minimizing user's response time and at achieving load sharing as well. We showed that most of the approaches for global distribution taking client proximity into account are not able to satisfy users' expectations. Instead, well-designed server redirection algorithms that exploit all components, that is activation, selection and location, enhance considerably the quality of Web services provided to end users in a geographical environment and have the ability to face sudden surges of load which are exacerbated in a geographical environment.
Section 8.2. Future Directions

To evaluate the performance results of the proposed algorithms and mechanisms taking network information into account, we also developed a simple model of the Internet infrastructure which provides a fair testbed for comparison.

- The analysis, classification, and qualitative evaluation of the existing architectures, routing mechanisms and dispatching algorithms for scalable Web-server systems. We broadly classified solutions into local and global ones, according to the topological network distribution of the server nodes comprising the Web system. Then, we analyzed in depth the two main categories focusing on the tradeoffs posed by the different approaches and the directions to develop more effective solutions. We examined how local and global architectures are able to satisfy the main system requirements, that is performance, scalability, and immediate applicability to the current Web infrastructure. We described and discussed for both locally and geographically Web systems the dispatching algorithms that can be used to select the server node most appropriate to provide the requested Web service and the routing mechanisms apt to direct client requests to the target server node. We proposed detailed taxonomies for both routing mechanisms and dispatching algorithms and discussed some representative research prototypes and commercial products within each class of architectures. Based on our investigation, we focused on geographically distributed Web systems, which have the capability to reduce network impact on user-perceived response time and to scale the distribution of Web content to a large number of clients.

8.2 Future Directions

In this thesis, we have addressed issues for designing scalable Web-server systems that support a large number of client requests. However, a number of challenges remain to be faced in future research to further scale up Web system capacity with service demand and network capacity.

First, the distribution of Web content and services in a global system has not been yet completely solved and needs further research. As secure transactions are becoming widespread used and Web services are rapidly extending to the provision of more sophisticated content, there is the need to investigate further location- and content-aware dispatching algorithms in a global environment. Although content distribution networks are evolving towards the delivery of other Web services such as
media streaming besides the distribution of static content, some services offered by a Web site, such as those involving secure transaction and dynamic content generation, cannot be easily outsourced to CDNs as they conflict with cacheability.

The wide deployment of globally distributed Web systems poses the problem of evaluating their performance in a real environment. While a variety of Web workload generators has been developed to evaluate the performance of single-node servers and can be adopted for locally distributed systems, very little effort has been made for evaluating and comparing alternative system architectures in a reproducible wide-area infrastructure.

In the near future, Web-server systems should also provide content accessibility which is related to the ability of the Web system to tailor multimedia content to the heterogeneous client devices with diverse computing powers, display capabilities and network bandwidths that are gaining accesses to the Internet, such as smart phones, PDAs, and handheld computers. The adaptation of content to address device heterogeneity as well as to personalize information according to user needs opens up new challenging and interesting problems. Indeed, Web content adaptation imposes on the Web system a considerable computational load which increases further that caused by the provision of current Web services. Therefore, dispatching algorithms should also consider the problem of distributing efficiently the content adaptation among the Web server nodes.

The extension of geographically Web systems to support quality of service (QoS) is another challenging problem. A few solutions that address the support for service differentiation have been recently proposed for Web clusters, in which layer-7 Web switches are the best candidate to deploy QoS-aware dispatching as their request classification can be based on all client information at the application level. However, the real challenge lies in the combination of network level and server level QoS mechanisms so to achieve an end-to-end quality of Web services.
Bibliography


