An adaptive admission control and load balancing algorithm for a QoS-aware Web system

A dissertation presented
by

Katja Gilly de la Sierra - LLamazares

to
The Department of Ciències Matemàtiques i Informàtica
in partial fulfilment of the requirements
for the degree of
Ph.D.
in the subject of
Computer Science

**Universitat de les
Illes Balears**

Universitat de les Illes Balears
Palma de Mallorca, Spain
September 2009

Dedicated to Lolo and to my parents

Acknowledgments

Now that this doctoral work is almost completed I am happy to say that it has been absolutely worth doing it. I have enjoyed most of the time I have been working on it over these last six years and I have learnt about many more things that I thought I was going to, like how to coherently write a paper, give a talk and work in a group of people, apart from staying focused and research in an area that is continuously changing.

It is also true that, at some points, it has been very difficult to keep going, but the support and useful comments of my supervisor has always helped me to get rid of the bad "research" moments and look forwards. First of all, I want to thank Dr. Carlos Juiz for the patience, support and friendly supervision he has given me. His invaluable advice and stimulation has made this thesis possible.

I also would like to thank Prof. Ramon Puigjaner, who has always given me kind advice and very interesting comments that helped me to improve this work. I also have to thank Salvador Alcaraz for the coffee discussions, comments and many jokes that helped me to keep my feet on the ground.

I am very grateful to Dr. Nigel Thomas, who invited me on two occasions to visit the School of Computing Science at Newcastle University for three months. This has permitted me, apart from improving my English and meeting amazing people, to speed up the progress of this thesis.

And finally I want to thank Lolo, my partner and guide during all these years for his constant and tireless support and love during the good and bad moments. I would also like to thank my parents who have always given their love and encouragement.

Thesis supervisor
Dr. Carlos Juiz García

Author
Katja Gilly de la Sierra - LLamazares

Abstract

The overload of the servers and the resulting decrease in the Quality of Service (QoS) and performance becomes more serious as the use of Web services grows. In order to avoid this, service providers use large distributed networks of servers to attend the requests of the increasing number of visits in popular sites.

The main objective of this thesis focuses on the design of an adaptive algorithm for admission control and content-aware load balancing for Web traffic. In order to set the context of this work, several reviews are included to introduce the reader in the background concepts of Web load balancing, admission control and the Internet traffic characteristics that may affect the good performance of a Web site.

The admission control and load balancing algorithm described in this thesis manages the distribution of traffic to a cluster of Web servers (or Web cluster) based on QoS requirements. The goal of the proposed scheduling algorithm is to avoid situations in which the system provides a lower performance than desired due to servers' congestion. This is achieved through the implementation of forecasting calculations. Obviously, the increase of the computational cost of the algorithm results in some overhead. This is the reason for designing an adaptive time slot scheduling that sets the execution times of the algorithm depending on the burstiness that is arriving to the system. Therefore, the predictive scheduling algorithm proposed includes an adaptive overhead control.

Once defined the scheduling of the algorithm, we design the admission control module based on throughput predictions. The results obtained by several throughput predictors are compared and one of them is selected to be included in our algorithm. The utilisation level that the Web servers will have in the near future is also forecasted and reserved for each service depending on the Service Level Agreement (SLA).

Our load balancing strategy is based on a classical policy. Hence, a comparison of several classical load balancing policies is also included in order to know which of them better fits our algorithm.

A simulation model has been designed to obtain the results presented in this thesis.

Director de tesis
Dr. Carlos Juiz García

Autor
Katja Gilly de la Sierra - LLamazares

Resumen

La sobrecarga de los servidores y la disminución resultante en la calidad de servicio (*QoS*) y rendimiento se agrava conforme la demanda de servicios Web aumenta. Con el fin de evitarlo, los proveedores de servicio utilizan redes distribuidas de servidores para atender el creciente número de visitas en sitios Web populares.

El objetivo principal de esta tesis se centra en el diseño de un algoritmo adaptativo de control de admisión y equilibrio de carga para tráfico Web. Con el fin de establecer el contexto de esta tesis, se realiza una descripción exhaustiva de los conceptos de distribución equilibrada de carga Web, control de admisión y las características del tráfico de Internet que pueden afectar el buen desempeño de un sitio Web, además de su estado del arte.

El algoritmo de control de admisión y equilibrio de carga descrito en esta tesis gestiona la distribución del tráfico en un clúster de servidores Web en base a los requisitos de calidad de servicio. El objetivo del algoritmo propuesto es evitar situaciones en las que el sitio Web proporciona un rendimiento más bajo de lo deseado debido a la congestión de servidores. Esto se consigue en base a cálculos de predicción. Obviamente, el aumento del coste computacional durante la ejecución del algoritmo supone una carga adicional en los recursos del sistema (*overhead*). Por esta razón, se ha diseñado una planificación adaptativa en la ejecución del algoritmo que varía en función de las ráfagas de tráfico Web que se detectan en el sistema (*burstiness*). Por lo tanto, el algoritmo propuesto incluye un control adaptativo del overhead.

Una vez definida la planificación del algoritmo, diseñamos el módulo de control de admisión basado en predicciones de productividad (*throughput*). Los resultados obtenidos por varios predictores de productividad se comparan y uno de ellos es seleccionado para ser incluido en nuestro algoritmo. El nivel de utilización de los servidores Web también se predice y se reserva para cada servicio en función de compromiso adquirido (*SLA*).

Nuestra estrategia de equilibrio de carga se basa en una política clásica de distribución. Por tanto, se incluye una comparación de varias de estas políticas ejecutadas en el algoritmo con el fin de saber cuál de ellas proporciona mejores resultados.

Se ha diseñado un modelo de simulación con el que se han obtenido los resultados que se presentan en esta tesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The increase of the use of Internet in all areas of common life has been very significant during last decade. Internet has indeed become an essential tool for our lives and it is evident that it has not stopped growing as it can still be improved in many aspects of its architecture. Many research groups worldwide continue their Internet research trying to speed up the communication protocols and improve the performance at the Web application level. Also new applications are being developed to be executed across Internet connections such as the Voice Over Internet Protocol (VOIP) telephony, on-line games and social networks.

In this thesis we focus on the Web server site by trying to offer a Web cluster solution that balances the load at the application layer and includes Quality of Service (QoS)-based admission control. We develop a content-aware load balancing algorithm for a cluster of Web servers that distributes the load based on different priorities in the services. We have considered it interesting to include a detailed survey of previous works in this area.

Overhead is controlled by an adaptive scheduling of our algorithm which permits it to be execute more frequently when intense workload and burstiness is detected in the Web site, and less frequently when there is a low risk of overload. Hence, the invocation times of our algorithm are adapted to the demand of the Web system.

An important aspect of this work focuses on the admission control of requests when the Web server side is overloaded. We deal with the resource allocation performed in the servers in order to guarantee the SLA contracted with the service provider. Based on a throughput predictor, our admission control algorithm permits a Web system to maintain its performance despite the Web requests arrival rate variations that occur due to the heavy-tailed nature of Internet traffic.

Surveys of burstiness detection and monitoring in Web traffic, and the admission control policies recently proposed are also included in order to describe the state-of-the-art.

In this introductory chapter, we introduce the main objectives of this work and its structure.

1.2 Main Objectives

The main objectives of this thesis are:

- To establish an up-to-date context of load balancing solutions during the last years in order to situate our work.
- To review the admission control proposals that exist in literature, analysing their main characteristics.
- To analyse the effects that burstiness in arrival rate may cause in a Web system and propose a burstiness factor that detects the variations in workload.
- To develop an adaptive time slot scheduling that allows a low overhead in the execution of the algorithm.
- To develop a QoS-aware admission control and content-aware load balancing algorithm that ensures the good performance of the Web system considering an intense arrival rate.
- To study the effects of different classical load balancing solutions included in our content-aware load balancing algorithm.
- To compare our algorithm with similar proposals.

1.3 Outline

Hence, we have divided this thesis into three main parts: *Background*, *Contribution* and *Conclusions, Appendix and References*. The first two parts help the reader in making a clear distinction among others work and our own research work and findings. The last part includes the general conclusions and open problems of this thesis.

Part I: Background In the first part we have included two different surveys. Firstly, we have studied and reviewed most of the Web load balancing policies that have been proposed recently in literature in Chapter 2. We have organised this survey by distinguishing between architecture and request distribution proposals. Secondly, a survey of burstiness detection and monitoring and the most significant admission controls solutions are included in Chapter 3.

Part II: Contribution The second part of this work includes two chapters. The design of a burstiness monitoring and detection mechanism that aims for low overhead is included in Chapter 4, and the description of the admission control and content-aware load balancing algorithm we propose is described in Chapter 5.

Part III: Conclusions, Appendix and References The third and last part includes the conclusions, an appendix, the bibliography and the acronyms list. The thesis conclusions and open problems are presented as Chapter 6. We have developed a simulation model to analyse the performance of the algorithm based on OPNET Modeler. The implementation details of our algorithm are added to this thesis as an appendix (Appendix A).

Part I

Background

Chapter 2

Web Traffic Load Balancing

This chapter introduces an up-to-date state-of-the-art in load balancing mechanisms that includes all possible classifications and focuses on the advantages of using load balancing solutions to increase the performance of the Web system. A general description of the Web load balancing solutions is included and organised by differentiating the Open Systems Interconnection (OSI) protocol stack layer the load balancing is based on. We also describe the most important request distributing policies that are included in literature.

2.1 Introduction

The main reason of the increasing popularity of server-based clusters, also called server farms, is due to the fact that Web applications must be able to run on multiple servers in order to accept an increasing number of users that demand Web content. In essence, load balancing is the ability to make several servers participate in the same service and do the same work, since the capacity of servers is finite. This implies important benefits such as *scalability*, *availability*, *manageability* and *security* of Web sites. First and foremost, load balancing improves *scalability* of an application or server cluster by distributing the load across multiple servers. Load balancing is also able to direct the traffic to alternate servers if a server or application fails. The ability of maintaining service unaffected during a predefined number of simultaneous failures is called *availability. Manageability* is improved by load balancing in several ways by allowing network and server administrators to move an application from one server to another easily. Last, but not least, load balancing solutions provide *security* improvement by protecting the server clusters against multiple forms of

Denial-of-Service (DoS) attacks.

The rest of this chapter is organised as follows:

- Section 2.2 briefly revises the load balancing classifications that appear in the literature.
- Section 2.3 introduces the concept of scalability as a main requirement of a modern Web-based system.
- Section 2.4 covers the load balancing scheduling solutions that are based on the TCP layer, also called layer-4 or content-blind load balancing solutions.
- Section 2.5 deals with the architectures that balance the load based on the application layer, also called content-based distribution.
- Finally, we include the summary of the chapter.

2.2 Load Balancing classifications

Several classifications of load balancing techniques exist in literature. In this section we sum up them.

- Depending on the taxonomy of the Web-server architectures, a distinction is made among the *local scale-out* and *global scale-out* approach [27]. The main difference consists of the geographical locations where the set of server nodes resides. In the global scale-out approach the nodes are located at different geographical locations while the nodes are in the same location in a local scale-out architecture. An example of a global scale-out organisation is the Content Distribution Networks (CDNs) [89, 21]. The local scale-out architectures are also called *locally distributed Web systems*.
- Another distinction can be made in this group depending on the visible Internet Protocol (IP) addresses the Web system presents to the client. If the IP addresses of the Web server nodes are visible to the clients, then we are referring to a *Distributed Web System*. When the only visible address to the client application is the Virtual IP (VIP) of the front-end of the Web system, then the architecture is a *Cluster-based Web System* (or Web cluster).

- Cardellini *et al.* in [28] proposed a classification depending on where the distribution decision is taken when routing a request to one server of a locally distributed Web-server system: *client-based*, *Domain Name System (DNS)-based*, *dispatcher-based* and *server-based*. In this work we mainly consider the dispatcher-based clusters, where a front-end of the Web system receives all incoming requests and distributes them among the servers. Client-based and DNS-based approaches are out of the scope of this thesis, however more information about client-based and DNS-based can be found in [28] and [26, 18, 80, 46], respectively.

- The classification proposed by Choi in [41] depends on the level the load balancing is applied to: hardware level (referred basically to commercial products), system software level, middleware software level and application software level. Despite being a very interesting classification, we do not consider it in this work as sometimes the proposals that appear in literature do not specify the level they are implemented in.

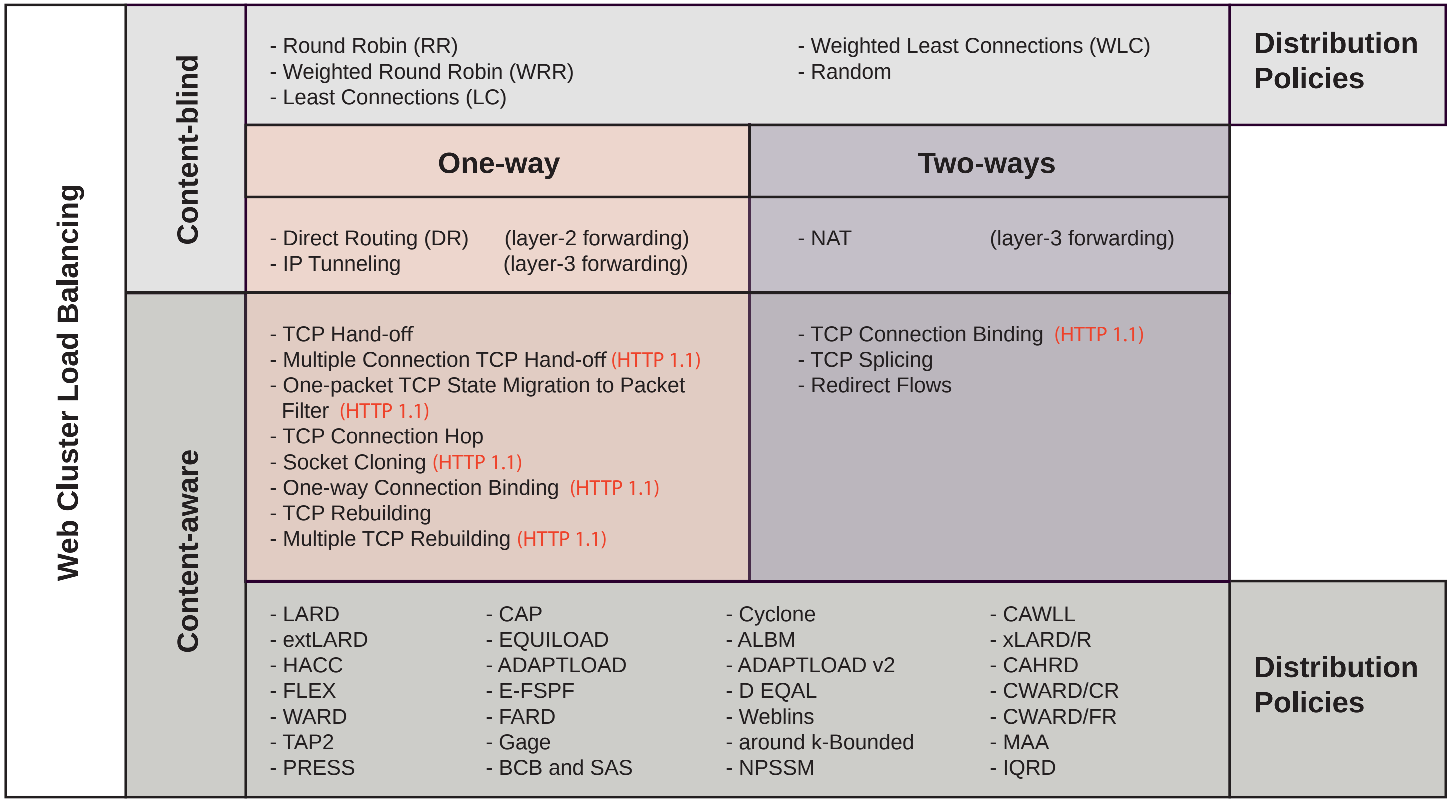
- Focusing on Web cluster load balancing, it is also quite usual to group the load balancing solutions depending on the OSI protocol stack layer the load balancer, also named Web switch or front-end, is based on. In the next sections we classify the load balancing solutions by distinguishing the layer they are based on.

- Also referring to Web cluster load balancing, there is an alternative classification depending on the return way of the data flow from the object server to the client. The response from the server can either go through the load balancer (*two-ways* architecture) or it can follow an alternative path direct to the client avoiding the load balancer (*one-way* architecture or single arm server load balancing). This last one mainly consists of a different path of traffic flow from the server to the client rather than passing through the load balancer in order to avoid a possible bottleneck in the load balancer.

In Figure 2.1 the classification we have used in this work is detailed.

2.3 Introducing scalability in load balancing

Among all the reasons for using load balancing solutions, we are going to focus on scalability as user demands placed on Web services continue to grow and Web server systems



Web Cluster Load Balancing

Content-blind

- Round Robin (RR)
- Weighted Round Robin (WRR)
- Least Connections (LC)
- Weighted Least Connections (WLC)
- Random

Distribution Policies

One-way

- Direct Routing (DR) (layer-2 forwarding)
- IP Tunneling (layer-3 forwarding)

Two-ways

- NAT (layer-3 forwarding)

Content-aware

- TCP Hand-off
- Multiple Connection TCP Hand-off (HTTP 1.1)
- One-packet TCP State Migration to Packet Filter (HTTP 1.1)
- TCP Connection Hop
- Socket Cloning (HTTP 1.1)
- One-way Connection Binding (HTTP 1.1)
- TCP Rebuilding
- Multiple TCP Rebuilding (HTTP 1.1)

- TCP Connection Binding (HTTP 1.1)
- TCP Splicing
- Redirect Flows

- LARD
- extLARD
- HACC
- FLEX
- WARD
- TAP2
- PRESS
- CAP
- EQUILOAD
- ADAPTLOAD
- E-FSPF
- FARD
- Gage
- BCB and SAS
- Cyclone
- ALBM
- ADAPTLOAD v2
- D EQAL
- Weblins
- around k-Bounded
- NPSSM
- CAWLL
- xLARD/R
- CAHRD
- CWARD/CR
- CWARD/FR
- MAA
- IQRD

Distribution Policies

Figure 2.1: Organisation of the Web load balancing solutions

are becoming more stressed than ever. There should therefore no longer be a limit on the performance of an application that is running on a single server. Load balancing avoids this bound by the ability of growing the number of servers that host the application.

Even though both network and server capacity have improved in recent years and new architectures have been developed, there are still some problems to be solved from the user point of view in terms of perceived response time. When a server is congested, the response times obtained by the user increase and this can lead to a lost sale operation when we are referring to an e-business site. Therefore response time continues to challenge server system and cluster related research.

We are going to focus in this thesis on the Web system infrastructure as it is the only component that can be under the direct control of the site administrator in a distributed network system as Internet. Other elements that compose the network such as DNS systems, backbones and routers are not controllable by a single organization and are out of the scope of this work.

The Web system architecture we consider consists of a collection of server computers that are locally distributed and interconnected through a high speed network. This architecture provides a single interface to the outside, hence, it can be seen as a single host. The users are not aware of the names and addresses of the servers that compose the Web architecture, they access the applications hosted in the system directing their requests to the VIP address corresponding to the device that acts as the front-end of the Web architecture. This kind of architecture is also named Cluster-based Web System [27].

2.4 Content-blind Load Balancing

The load balancing solutions covered in this section are called content-blind because the load balancer is unaware of the application information contained in incoming requests. Load balancers that perform content blind routing are normally referred as layer-4 load balancers. The selection of the target server that is going to attend the request is done based on the information contained in the TCP SYN packet at the load balancer. The OSI layer used to forward the incoming packet to the target server can be either the link or the network layer.

Table 2.1 summarises all the solutions covered in this section.

We have divided this section into three subsections:

Table 2.1: Content-blind Load Balancing Solutions

	Layer-2 forwarding	Layer-3 forwarding
Two-ways		- Network Address Translation
One-way	- Direct Routing	- IP Tunneling

- Subsections 2.4.1 and 2.4.2 detail the techniques that balance the load depending on the OSI layer (2 or 3, respectively) the front-end uses to forward the packets to the servers.

- Subsection 2.4.3 details the scheduling policies that can be applied in a content-blind load balancing solution.

2.4.1 Layer-2 Forwarding

Layer-2 forwarding (or *bridging Server Load Balancing (SLB)*) is the most simplistic solution for load balancing and can be considered when all interfaces of the Web system architecture are in the same Virtual LAN (VLAN) and IP network, including the client-side router. There is no need to alter the topology of the network nor redefine the IP addressing map. It is only necessary to include a bridge device between the client-side router and the server's side.

The client basically establishes a TCP connection with the server that is going to attend its request through the VIP of the Web site. The function of the front-end device, that acts as the load balancer, is to select the server and translate the destination Media Access Control (MAC) address to leave no evidence there is an intermediary device in the communication [128].

Figure 2.2 illustrates an example of layer-2 forwarding, detailing the role of the load balancer that has to re-write the layer-2 destination address to the MAC address of the selected Web server and then forward the incoming request to that server. Reverse translations are performed when the response is sent back to the Web client. It is important to notice that the load balancer does not change the IP address of the incoming request because all the devices in the IP subnet (the load balancer and the servers) share the same IP address. Hence, there is no need to change the network information of the packet and consequently, there is also no need to recompute the IP checksum, which means less overhead. It is important to disable the Address Resolution Protocol (ARP) when using layer-2

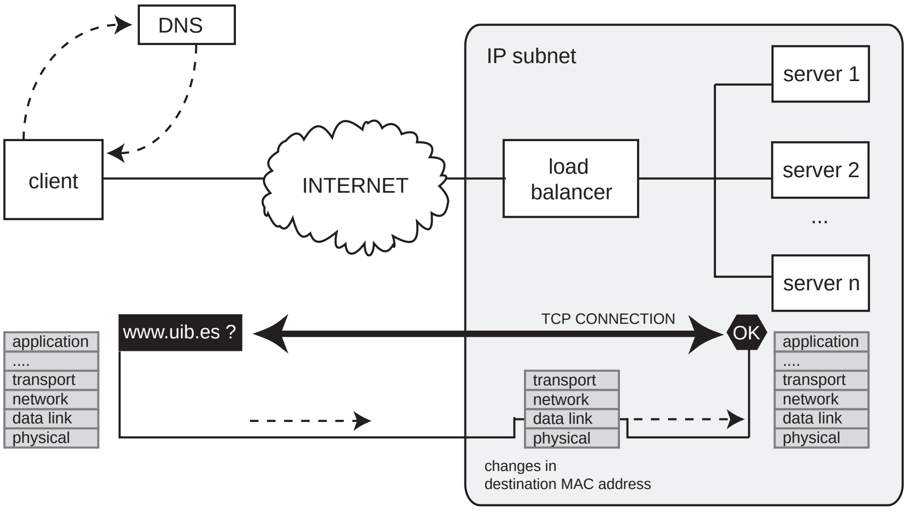


Figure 2.2: Example of a two-ways layer-2 forwarding implementation

forwarding to avoid a possible collision because the same IP address has been assigned to all the nodes of the system. A layer-2 load balancer uses the MAC address available in the data link layer information to determine the output interface port for that packet [88, 56].

Layer-2 forwarding has been widely used in commercial solutions in its one-way architecture version, and is normally named *layer-4 switching with layer-2 packet forward* (L4/2) [37, 73, 120] or Direct Routing (DR) [132, 149]. As the same IP address has been assigned to all the devices of the subnet, the outgoing packets can be sent directly from the server to the clients without going through the load balancer. Figure 2.4 shows the TCP connection establishment with DR technique.

Some pioneering prototype implementations of L4/2 load balancing were ONE-IP [49] and LSMAC [56]. Nortel Networks also consider this mechanism in their actual Nortel Application Switches [105]. Linux Virtual Server (LVS) is a layer-4 load balancing solution that is included in an open source project that was started by Zhang in 1998 [121, 149]. It can be configured to support DR, and also other forwarding techniques that are included in the next subsection.

2.4.2 Layer-3 Forwarding

Layer-3 forwarding (also called *routing SLB*) differs from bridging SLB in that the client-side router can be in different VLANs and IP subnets in the Web system architecture. In

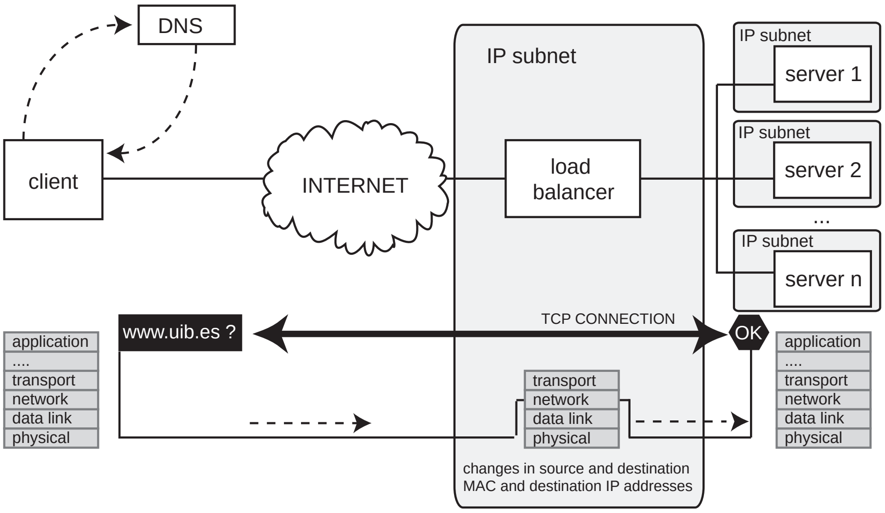


Figure 2.3: Example of a two-ways layer-3 forwarding implementation

fact, in this case, the load balancer of the Web system architecture is now routing rather than bridging the frames between the client and the server. Figure 2.3 illustrates an example of layer-3 forwarding and shows that a layer-3 load balancer de-encapsulates a packet up to the network layer to determine where to send the packet.

Two forwarding techniques have been implemented in a dispatcher-based web cluster with layer-3 routing: *Network Address Translation (NAT)* and *IP Tunneling (IPTun)*. NAT is the simplest technique and consists of rewriting the layer-3 destination address of the incoming packet to the IP address of the real server selected by the load balancer. The example of Figure 2.3 represents a NAT implementation. While NAT is implemented in a two-ways architecture, IPTun is implemented in a one-way the architecture. This means that the response from the chosen Web server goes directly to the client. IPTun consists of the encapsulation of IP datagrams within IP datagrams with the source and destination IP address specifying the VIP address of the system and the target server IP address, respectively. More information about NAT and IPTun can be found in [142, 132, 27]. Figure 2.4 also shows the TCP connection establishment when using NAT and IPTun.

Layer-3 forwarding solution based on NAT have also been classified as *layer-4 switching with layer-3 packet forwarding* (L4/3) by some authors [37, 73, 120].

The main disadvantage of this approach is the fact that the load balancer can become the bottleneck of the system as the workload increases. This is due to the overhead of

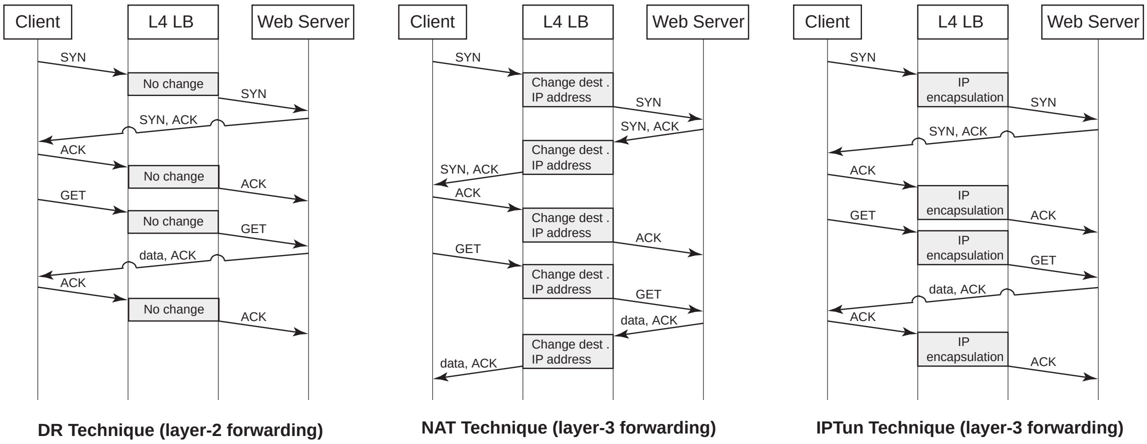


Figure 2.4: TCP connection establishment when using layer-2 and 3 forwarding techniques

recomputing the IP checksums for every packet that have to go through the load balancer in both ways.

Most of the commercial content-aware switches that are nowadays on the market also provide NAT forwarding, that is the case of the CSS 11500 Series Content Services Switch of Cisco System, Inc. [1, 43], ServerIron layer 4-7 switches of Foundry Networks [104] and Nortel Layer 2/7 Gigabit Ethernet Switch Module (GbESM) for IBM BladeCenter [105, 79], for example. The Linux software framework, LVS [121], also can be configured to support NAT and IPTun but the most efficient mechanism is DR [95, 40]. Microsoft also implements a layer-3 forwarding mechanism named Network Load Balancing (NLB) that is included in the Windows Server 2003 Family [32].

2.4.3 Content-blind Request Distribution Policies

Content-blind load balancing permits the front-end device to be aware of the TCP connections among the clients and the servers. Hence, the load balancer dispatches the requests according to the IP address and the TCP port. There are several load balancing scheduling policies that are normally used by content-blind load balancers. Some examples of these policies are:

- *Round Robin (RR)* Algorithm : the TCP connections are assigned on a RR basis, with the first connection going to server 1, the second to server 2, and so on. As the connections are assigned sequentially among the servers, each server receives the same number of connections over time independently of how fast it is able to process them.

For this reason, RR is one of the best distributing methods for homogeneous servers but, unless used with a per-server weighting, it is less effective in environments where the servers are heterogeneous.

- *Weighted Round Robin (WRR)* Algorithm : the traffic will be assigned to the servers according to their configured relative capacities, in the case that the servers are heterogeneous. The administrator specifies the percentage of traffic to be directed to each of the servers.

- *Least Connection (LC)* Algorithm : connections are assigned to the server with the least number of connections. This is a dynamic scheduling algorithm as the load balancer needs to count the number of connections that are established among the clients and each Web server in the cluster.

- *Weighted Least-Connection (WLC)* Algorithm : similar to LC, in this algorithm apart from counting the number of connections, a weight assigned to each server is also considered. The servers with higher weight will receive a larger percentage of connections than the rest of the servers.

- *Least Loaded (LL)* Algorithm : the dispatcher assigns the next request to the server that has the lowest load. In this case an agent on the server keeps the load balancer updated on the server utilization and capacity. Connections are assigned to the server having the most spare capacity. It is also called baseline algorithm.

- *Random* Server Selection : connections are assigned uniformly among the servers but not in a deterministic sequence.

2.5 Content-aware Load Balancing

A content-aware load balancer works at application layer. This means that the load balancer is aware of the application content of the incoming request. The TCP connection must be established first among the client and the front-end of the Web system, to then receive the HyperText Transfer Protocol (HTTP) request and analyse the content of it (see Figure 2.5). This makes the content-aware routing more specific to applications that can offer differentiated services, but also more complex than the content-blind approach.

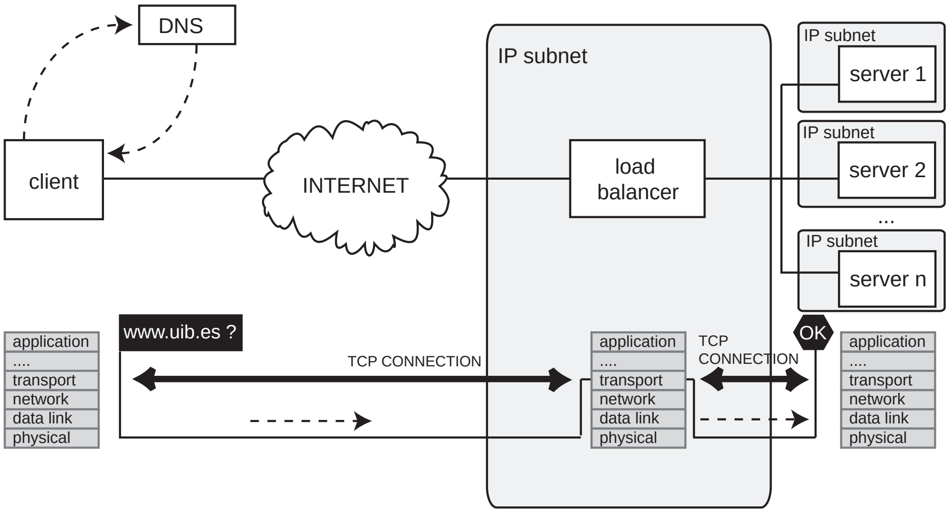


Figure 2.5: Example of layer-7 two-ways load balancing

Despite the fact that some content-aware load balancing commercial solutions have been implemented, it is a subject that is still being investigated. In this section we are going to analyse the most recent content-aware solutions described in literature. Some works prove that using the content of requests and loading information from the servers, more flexible and intelligent distributing algorithms can be developed [9, 12, 27, 31, 38, 93, 107, 110, 125].

Let us introduce the fact that HTTP/1.1 permits persistent (keep-alive) connections [55]. This means that several HTTP requests from a client can go through the same TCP connection. Hence, this causes a reduction in response time, server overhead and network overhead of HTTP [103]. In order to take advantage of these benefits, some TCP modifications have been developed to permit an HTTP request granularity in the content-aware load balancing, instead of a connection granularity. These modifications depend on the one-way or two-way architecture of the Web cluster.

Table 2.2 summarises all the solutions covered in this section, indicating if they include request granularity when using the HTTP/1.1 protocol.

Let us divide then this section into one-way and two-ways architectures as it is crucial to describe their TCP behaviour to know how the load balancing is done. Hence, this section contains the following subsections:

- Subsections 2.5.1 and 2.5.2 review the load balancing proposals depending on the return path of responses from the Web severs.

Table 2.2: Content-aware Load Balancing Solutions

	HTTP/1.0	HTTP/1.1
Two-ways	- TCP Splicing [98] - Redirect Flows [47]	- TCP Connection Binding [141]
One-way	- TCP Hand-off [77] - TCP Connection Hop [4] - Socket Cloning [124] - TCP Rebuilding [94]	- Multiple Connection TCP Hand-off [12] - One-packet TCP State Migration to Packet Filter [93] - Multiple TCP Rebuilding [95] - One-way Connection Binding [97]

- Subsection 2.5.3 describes the scheduling policies that can be applied in a content-aware load balancing solution, including an up-to-date survey of recent proposals in literature.

2.5.1 Two-ways Architectures

This subsection introduces three mechanisms to route the requests from the load balancer to the target Web server in a two-ways architecture, that are: *TCP Connection Binding* [141], *TCP Splicing* [98] and *Redirect Flows* [47].

TCP Connection Binding

Yang and Luo proposed TCP Connection Binding in [141]. It is also called *TCP Gateway* by other authors [45, 27], *Relaying front-end* in [12] or *Relaying with Packet Rewriting* in [125], and basically consists of maintaining two TCP connections: one between the client and the load balancer, and a second one between the load balancer and the Web server. Before receiving any request, the load balancer establishes a persistent connection with each Web server. When the load balancer receives a request from a client, one of these pre-established TCP connections is used to transfer the request to the selected target server. The main advantage of this proposal is that it permits a content-based distribution at the granularity of individual requests because the persistent connections between the load balancer and the back-end servers do not depend on the incoming traffic [12]. In a later work [96], the authors improve the request distribution and the reliability of the TCP Connection

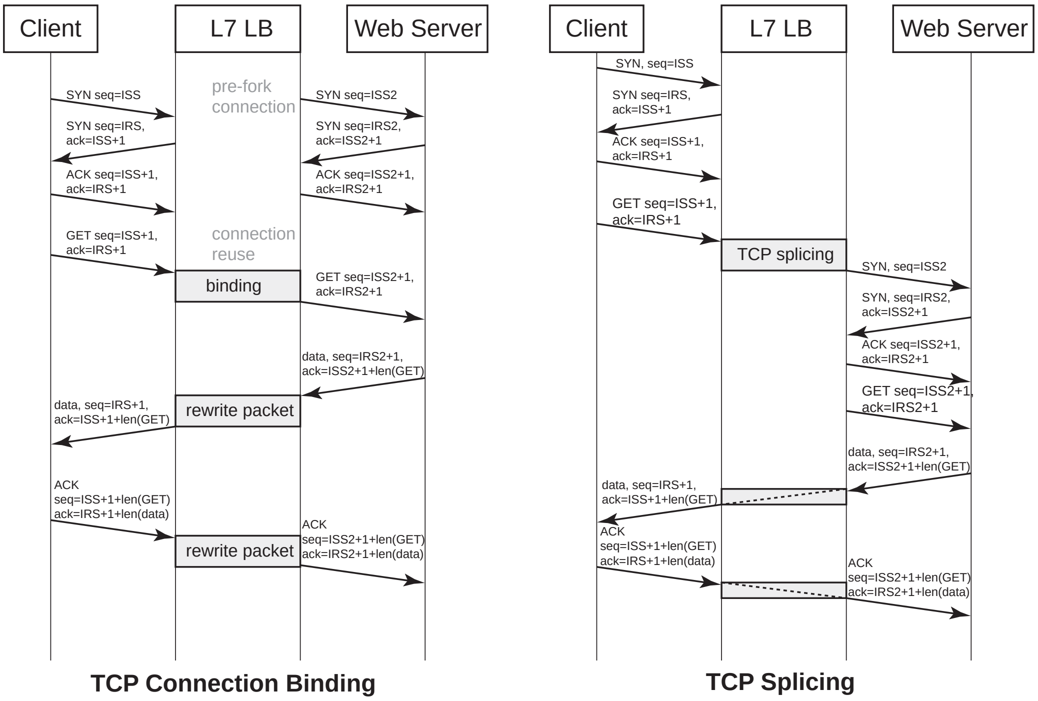


Figure 2.6: Layer-7 load balancing techniques in a two-ways architecture: TCP Connection Binding and TCP Splicing

Binding mechanism, and also include a Java implementation of their proposal. The main problem of this approach is that the packets need to be analysed up to their application layer information when passing through the load balancer. This implies an overhead that can be avoided by the other mechanisms detailed below.

Figure 2.6 shows the TCP Connection Binding procedure, detailing Initial Send Sequence (ISS) number of the client and the Initial Receive Sequence (IRS) number of the load balancer used in the three-way handshake [50] of the client connection with the load balancer. Different numbers, ISS2 and IRS2, are used in the pre-forked connection between the load balancer and the Web server. All these numbers do not need to have any relation at all as the load balancer is responsible for changing the ISS and IRS TCP header field numbers depending on which connection it is using to transmit the packets.

IBM Network Dispatcher [78] was an example of a TCP Connection Binding commercial implementation, but was withdrawn from marketing some years ago.

TCP Splicing

The second mechanism, TCP Splicing, was proposed by Maltz and Bhagwat in [98]. This proposal is similar to TCP Connection Binding, but the performance is improved due to the fact that the TCP client connection and the TCP server connection with the load balancer are spliced together (at the TCP layer) and all the work can be carried out directly by the operative system forwarding the data at the IP level. Figure 2.6 also shows an example of TCP Splicing detailing the ISS and IRS TCP header field numbers of both connections.

Some software and hardware implementation designs of Web switches that use TCP Splicing have been proposed. Cohen *et al.* in [45] describe the implementation details of a Web switch based on the Linux Operative System (OS). Rosu and Rosu in [116] propose a socket-level implementation of TCP Splicing and compare it to an IP-level implementation on AIX RS/6000 machines concluding that the socket-level implementations provides more flexibility and improves the transfer rates. Other authors like Apostolopoulos *et al.* [10], Zhao *et al.* [151] and Kachris and Vassiliadis [81] propose a hardware design for a Web switch based on a PowerPC 603e processor, an Intel IXP2400 network processor and a multi-processor reconfigurable logic platform (Xilinx Virtex 4 FPGA), respectively.

Some authors have included modifications to the TCP Splicing mechanism in their research studies. This the case of Marwah *et al.* [99]. They propose some enhancements that include the possibility to split the splice in the Web server and then send the responses directly to the clients. This permits the implementation of a one-way architecture when using TCP Splicing. The authors also propose in [99] to resplice an already established TCP connection with one server to attend an incoming request in another (possibly more appropriate) target server, which permits request granularity in the load balancing instead of TCP connection granularity when using HTTP/1.1. The authors also provide a prototype implementation in Linux that include some of the enhancements to the TCP Splicing mechanism they propose.

An extension of TCP Splicing is documented by Chang *et al.* in [33]. It consists of having pre-forked connections between the load balancer and the Web Servers that are spliced to the connections between the clients and the load balancer when a request is received in the system. This approach is very similar to the TCP Connection Binding approach proposed in [141].

Kobayashi and Murase in [86] deal with persistent TCP connections trying to provide a request granularity in the load balancing. They propose an asymmetric TCP Splicing that permits it to receive pipelined HTTP requests through a TCP connection. After analysing the application layer of the requests, their proposal change the target server that is serving the requests coming through that TCP connection for a more appropriated server, in case it is necessary.

Similar mechanisms to TCP Splicing have been implemented in some commercial content-aware solutions such as the Cisco CSS 11500 Series Content Services Switch [1, 43], F5's BIG-IP [53], Foundry's ServerIron layer 4-7 switches [104], Radware OnDemand switches [112] and Nortel Layer 2/7 Gigabit Ethernet Switch Module (GbESM) for IBM BladeCenter [105, 79]. Also a Linux framework has been developed to support TCP Splicing named Linux Layer7 switching (L7SW) [127].

Redirect Flows

The third and last two-ways mechanism, Redirect Flows developed by Colby *et al.* [47], is very similar to the TCP Splicing approach but based on the NAT architecture [40, 95]. It was a proprietary mechanism of Arrowpoint Communications Inc., a company that was acquired by Cisco Systems Inc. [1] in year 2000.

2.5.2 One-way Architectures

The main disadvantage of the two-ways architecture proposals is that response data must be forwarded by the load balancer, that may become the bottleneck of the system when high volume of traffic needs to be processed.

This subsection describes the approaches that permit the Web servers to return responses directly to clients. Some mechanisms have been proposed to route the requests from a target Web server to clients in a one-way architecture: *TCP Hand-off* [77], *One-packet TCP State Migration to Packet Filter* [93], *TCP Connection Hop* [4], *Socket Cloning* [124] , *One-way Connection Binding* [97] and *TCP Rebuilding* [95] .

TCP Hand-off

The most popular solution for a layer-7 one-way Web cluster architecture is TCP Hand-off, that was proposed by Hunt *et al.* in [77]. It requires some modifications in the OS of the load balancer and the Web servers because once the TCP connection between the

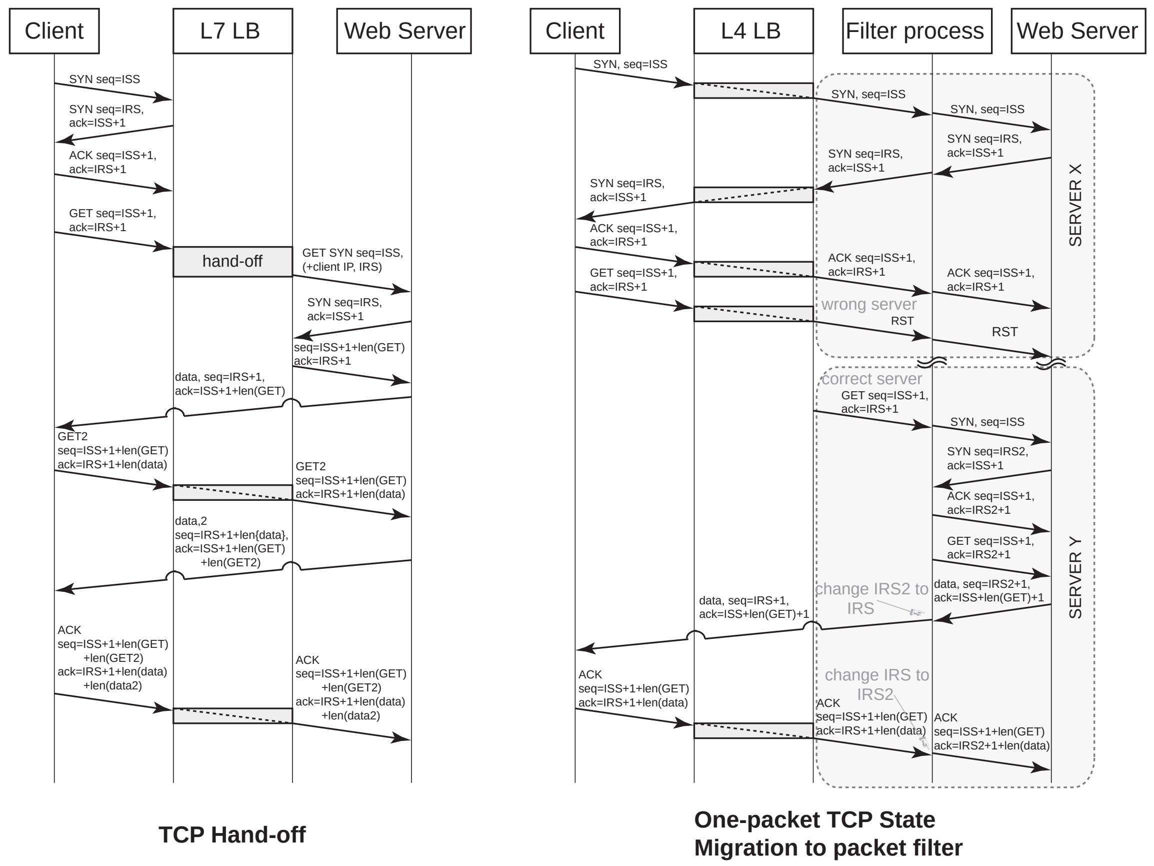

Client
L7 LB
Web Server
SYN seq=ISS
SYN seq=IRS, ack=ISS+1
ACK seq=ISS+1, ack=IRS+1
GET seq=ISS+1, ack=IRS+1
hand-off
GET SYN seq=ISS, (+client IP, IRS)
SYN seq=IRS, ack=ISS+1
seq=ISS+1+len(GET) ack=IRS+1
data, seq=IRS+1, ack=ISS+1+len(GET)
GET2 seq=ISS+1+len(GET) ack=IRS+1+len(data)
GET2 seq=ISS+1+len(GET) ack=IRS+1+len(data)
data,2 seq=IRS+1+len{data}, ack=ISS+1+len(GET) +len(GET2)
ACK seq=ISS+1+len(GET) +len(GET2) ack=IRS+1+len(data) +len(data2)
ACK seq=ISS+1+len(GET) +len(GET2) ack=IRS+1+len(data) +len(data2)
TCP Hand-off
Client
L4 LB
Filter process
Web Server
SYN, seq=ISS
SYN, seq=ISS
SYN, seq=ISS
SYN seq=IRS, ack=ISS+1
SYN seq=IRS, ack=ISS+1
SYN seq=IRS, ack=ISS+1
SERVER X
ACK seq=ISS+1, ack=IRS+1
ACK seq=ISS+1, ack=IRS+1
GET seq=ISS+1, ack=IRS+1
ACK seq=ISS+1, ack=IRS+1
wrong server
RST
RST
correct server
GET seq=ISS+1, ack=IRS+1
SYN, seq=ISS
SYN seq=IRS2, ack=ISS+1
ACK seq=ISS+1, ack=IRS2+1
GET seq=ISS+1, ack=IRS2+1
data, seq=IRS2+1, ack=ISS+len(GET)+1
SERVER Y
change IRS2 to IRS
data, seq=IRS+1, ack=ISS+len(GET)+1
change IRS to IRS2
ACK seq=ISS+1+len(GET) ack=IRS+1+len(data)
ACK seq=ISS+1+len(GET) ack=IRS+1+len(data)
ACK seq=ISS+1+len(GET) ack=IRS2+1+len(data)
One-packet TCP State Migration to packet filter

Figure 2.7: Content-aware load balancing techniques in a one-way architecture: TCP Hand-off and One-packet TCP State Migration to Packet Filter

client and the load balancer is established, the load balancer's end point of the connection is transferred to the selected server, as illustrated in Figure 2.7. The IRS number of the first connection and the client's IP address are sent to the Web server because the handed-off connection between the load balancer and the Web server has to be a "copy" of the client's connection in order to permit the Web server to send the responses directly to the client. Hence, the same ISS and IRS numbers have to be used in both connections. Pai *et al.* in [107] introduce some modifications to Hunt's Hand-off and apply it to their request distribution algorithm (Locality-Aware Request Distribution (LARD)), that is covered in the next subsection.

Aron *et al.* include some modifications to the TCP Hand-off mechanism to permit a granularity of individual requests when using HTTP/1.1 persistent connections in [12]. In this work, they propose the *Multiple Connection TCP Hand-off* and *Back-end Request*

Forwarding mechanisms. Multiple Connection TCP Hand-off basically permits pipelined incoming requests to be attended by different Web servers by migrating the connection between the servers. Back-end Request Forwarding avoids the overhead introduced by Multiple TCP Hand-off by permitting the redirection of requests from one server to another after a decision of the load balancer. Simple Hand-off is used when a new TCP connection request arrives to the load balancer. In [12], the authors study the performance of both methods and conclude that the Back-end Forwarding has a better performance for small response sizes while Multiple TCP Hand-off is better for large responses.

In a later work [14], Aron *et al.* compare the performance of TCP Splicing and TCP Hand-off and show how TCP Hand-off is more scalable with the number of Web servers in the cluster [14]. They also affirm in this work that the TCP Hand-off mechanism has a limited scalability of cluster sizes up to four Web servers, despite this mechanism implements a one-way architecture. They propose in [14] an alternative distributed load balancing solution with a layer-4 front-end, where a centralised node called the dispatcher takes the load balancing decisions and the overhead is carried out by the several distributor nodes that are connected in the network as they distribute the client requests to the selected Web servers by handing off the connections.

Similar content-aware mechanisms based on a content-blind Web switch as [14] are proposed in [109, 83, 38, 144]. *Knowledgeable Node Initiated TCP Splicing (KNITS)* [109] is a proposal of Papathanasiou and Hensbergen that uses NAT to forward the client's requests from the front-end to the Web servers, and multiple hand-off mechanisms are used when a session has to be migrated to another server, splicing the connection between the client and the front-end. The authors implement a prototype within the Linux Netfilter infrastructure. The Kerdlapanan and Khunkitti proposal [83] distributes the incoming requests to the servers by multicast through a layer-2 or 3 front-end (when the first SYN datagram is received). One of the servers of the cluster establishes the TCP connection with the client based on a hash function. The distribution decision is done by the servers, after being aware of the content requested. If the most appropriate server for the request is not the "connected server", then the connection needs to be transferred by using the Hand-off mechanism. Cherkasova and Karlsson describe a solution in [38] that reduces the forwarding overhead of the Multiple Hand-off in their strategy named Workload-Aware Request Distribution (WARD)(described in Subsection 2.5.3). Zeng-Kai *et al.* also propose in [144] a layer-7 load balancing distribution based on a layer-4 front-end. The content-aware distribution is done by the distributors that are located in the Web servers. The authors compare the performance of their proposal to LARD and WARD [107, 38], respectively, and show its benefits.

Considering request granularity on HTTP/1.1, Kokku *et al.* suggest another mechanism based on Hand-off, named *Half-pipe Anchoring*, that permits the distribution among different servers of individual requests that come through the same TCP connection [87]. Their proposal is based on the consideration of a TCP connection as two unidirectional half-pipes, one from the Web cluster to the client (data pipe) and one from the client to the Web cluster (control pipe). The selection of a different server to attend a pipelined request is possible by changing the origin of the data pipe of a connection. The authors extend the TCP header by including more TCP options in the message structure and implement a prototype in the Linux kernel. Comparison results between this proposal and KNITS [109] show that Half-pipe Anchoring has a maximum of 25% of the overhead produced by KNITS. Another implementation of TCP Hand-off, this one based on STREAMS-based TCP/IP is presented in [130]. This solution does not require any modification on the TCP/IP code.

TCP Hand-off has been implemented in Linux in the TCPHA project [5]. In this subproject of LVS, a layer-7 kernel level is implemented for Linux. Persistent HTTP connections are also supported in this one-way content-aware load balancing solution.

One-packet TCP State Migration to Packet Filter

The second mechanism, One-packet TCP State Migration to Packet Filter, was proposed by Lin *et al.* in [93]. They include a Packet Filter process in each Web server to intercept the connection from the load balancer without modifying the OS kernel and implement their proposal in LVS [121]. In order to provide more scalability, a pre-allocation scheme is also proposed in this work. It establishes a TCP connection with a Web server when receiving the first SYN (acting as a content-blind load balancer). Once the HTTP request is received, the load balancer determines if the selected server is correct or not. If the server is the right one, then the request is attended. If it is wrong (possibly because the content the client is asking for is not stored in the selected server) then the load balancer redirects the request to a more appropriate server, after a three-way handshake connection establishment as illustrated in Figure 2.7. A RST packet is sent to the previous server in order to keep silent this connection. TCP persistent connections are also supported. When a connection

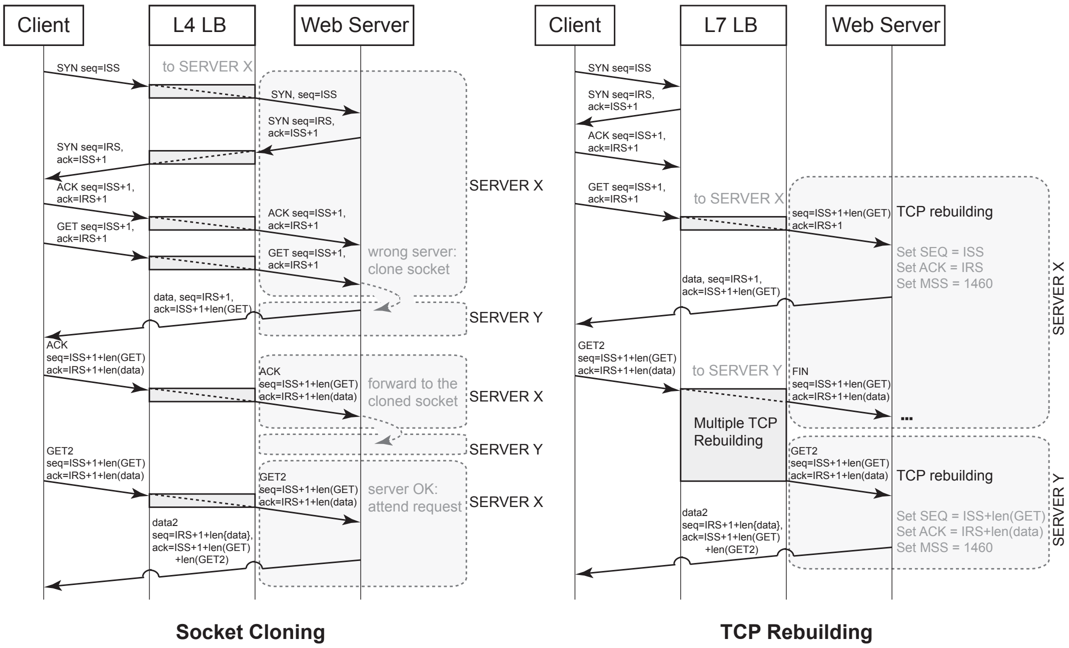


Figure 2.8: Content-aware load balancing techniques in a one-way architecture: Socket Cloning and TCP Rebuilding

that was previously used is needed again, the three-way handshake can be avoided because the connection was previously established and it is in a waiting status after the RST.

TCP Connection Hop

TCP Connection Hop is a proprietary solution of Resonate [4] and the core of their Resonate Central Dispatch. A software component called Resonate Exchange Protocol (RXP) has been developed to provide several useful functions [113]. It is installed in all the components of the cluster. RXP establishes the TCP connection with the client and performs the load balancing by transferring the connection to the selected Web server. The mechanism used to transfer the connection to the server is based on the TCP connection migration protocol proposed in [25] and is proprietary of Resonate [4]. It is a similar mechanism to TCP Hand-off and is also referred to as TCP State Migration in [97].

Socket Cloning

Sit *et al.* propose a mechanism to redirect the processing of a client's request from one server to another by migrating an open socket. This mechanism is named Socket Cloning and is proposed in [124]. The load balancer the authors consider in this work is a layer-4 switching with layer-2 packet forwarding, that makes the distribution decision when the client's SYN packet is received (see Figure 2.8). If the decision was not correct and the request has to be attended by another server, then the socket is cloned to it. Hence, Socket Cloning supports HTTP/1.1 persistent connections. A synchronisation process between the original and the cloned socket needs to be performed in order to update the sequence and ACK numbers after a response packet is sent. This synchronisation process is done by the packet router (that is an additional layer in all the nodes' architecture proposed by the authors), therefore it does not involve additional inter-node communication. The implementation of Socket Cloning requires some modifications to the OS kernel.

Takahashi *et al.* in [129] propose a similar solution. They use a layer-3 NAT forwarding mechanism to perform a layer-7 load balancing that provides TCP session redirection between the Web servers that compose the Web cluster, named *TCP-migration* by the authors. They physically separate the packet forwarding and request dispatching mechanisms. An implementation in Linux kernel is also described and its results reported in this paper.

Very similar mechanisms have been designed to avoid a failover [126, 148]. Snoeren *et al.* in [126] implement an architecture in Linux that provides a connection failover mechanism. The dispatching decision is taken in the Web servers, which can migrate the connection with the client to another more appropriate server. Zang *et al.* design a transparent TCP connection failover mechanism to recover connections that were lost due to a failure in the server that is attending Web requests [148]. They propose that each connection in the Web cluster system has to be visible by at least two servers: the primary server and one or more backup servers that will act in case of a failure of the primary. In both cases [126, 148], the backup servers need a constant synchronisation with the primary server.

One-way Connection Binding

The One-way Connection Binding mechanism has some similarities with TCP Connection Binding described in previous section (see Figure 2.6). It is introduced by Luo *et al.* in [97] and solves the problem of persistent connections in content-aware routing mechanisms by establishing long-lived connections at the server-side. When a request arrives to the front-end of the system, the client-side connection is bound to an appropriate server's connection and the request sent to the server to be attended. The server-side connection is reused by other requests avoiding the connection migration and termination that would require the execution of the slow-start mechanism and, consequently, the loss of performance that would involve. The authors compare the performance of their mechanism to the TCP Hand-off.

TCP Rebuilding

The fifth and last mechanism is TCP Rebuilding which was proposed by Liu *et al.* in [94, 95] to be implemented in a LVS-Content-Aware Dispatching (CAD) platform (this is described in Subsection 2.5.3). Figure 2.8 details its procedure: when the connection between the client and the load balancer is established and the HTTP request has arrived to the load balancer, it is sent to the selected Web server which starts rebuilding the TCP connection without the need of interchanging any more packets. Thus, the Web server has to guess the sequence and ACK number used in client's connection. The Maximum Segment Size (MSS) is set to the standard value of 1460. In order to handle HTTP/1.1 persistent connections, the authors propose the *multiple TCP Rebuilding* mechanism in [95]. As LVS cannot perform content-aware distribution, they have introduced a fast TCP module handshaking in the IP-layer of the front-end so that it can establish the connection with client in order to know the request's type.

2.5.3 Content-aware Request Distribution Policies

After studying the content-aware load balancing architectures, let us analyse in more detail the content-aware request distribution policies that have been proposed in recent literature. We have considered a distinction between the policies that try to exploit the cache of the system, named locality-aware solutions, and the ones that do not include the cache performance in the evaluation, named non locality-aware solutions. Table 2.3 sums up, in chronological order, the policies that are described in this subsection including some of their most important characteristics. The year of the publication of the paper that describes the distribution policy is detailed in the first column. The second column of this table indicates whether the policy has the locality-awareness feature. The third column details if dynamically generated Web content is considered in the workload. The fourth column deals with HTTP/1.1 persistent connections and request granularity. The fifth and sixth indicate the one- or two-ways architecture, and whether the Web switch is content- or blind-aware, respectively. Finally the seventh column indicates the load balancing mechanism used in the policy.

Locality-aware solutions

Several locality-aware solutions have been proposed as distribution policies in a content-aware load balancing design. The aim of these policies is to exploit the cache locality of the Web servers by sending the requests that ask for a determined Web page to a server that is likely to have it in its cache module. As it can be observed in Table 2.3, most of the solutions that are in literature between the years 1998-2001 improve the performance of the Web cluster increasing the number of cache hits in the Web server nodes.

Locality-aware policies deal with a working set that normally consists of static Web pages. Thus, in order to obtain an increase in the performance by improving cache hit rates and if the working set is too large to fit in one server's cache, it is divided among the nodes of the cluster. This is the idea that Pai *et al.* introduced in their distribution strategy named *LARD* [107]. They define a set of nodes to serve a target file, and modify the set dynamically depending on the number of active connections of the nodes. The content-aware load balancer sends the request to the least loaded node of the set. They implement TCP Hand-off to send the connections from the front-end to the selected back-end node. An extension of LARD (*extLARD*) is introduced in [12] which permits LARD to deal with persistent connections. Also LARD is used to evaluate a prototype implementation that includes a resource management facility for cluster based Web servers named *cluster reserves* [13]. This facility permits the performance isolation of the different server resources in order to effectively reserve them for different classes of service.

As an alternative to using the TCP Hand-off, which introduces an important overhead in the Web Switch [14], a DNS infrastructure is used by Cherkasova *et al.* in [37, 38]. *FLEX* is defined in [37] as a two-ways locality-aware solution that does not require any modification in the protocol nor special hardware support. It is mainly based on a DNS infrastructure that allocates different sites to the nodes of the cluster. Based on the access rates and the size of the files, the working set is dynamically partitioned among the servers equally. Periodical monitoring permits detection of changes in the access pattern and repartition of the working set . In a later work [38], the authors propose *WARD*, a locality-aware distribution that defines a core of files that contains the most frequently accessed files.

These files can be served by any node in the cluster while the rest of the working set is divided among all the nodes. In this case, Multiple TCP Hand-off is used when a request has to be sent from one server to another. The optimal size of the core is determined by an algorithm that considers the workload, the Random Access Memory (RAM) of the cluster and the overhead of the Hand-off operation and disk access.

There are other proposals that also includes a set of the most accessed files to be cached in all the servers [110, 125, 54, 40]. A prefetching policy named *Time and Access Probability-based Prefetch (TAP^2)* is proposed by Park *et al.* in [110]. It predicts the next Web requests the clients are going to ask by considering the costs of prefetching and the probability of the Web object to be requested, and if worth, it prefetches the requested objects from local disks. A client session is assigned to a back-end server by a RR policy, and a TCP connection between them remains during all the session. Sit *et al.* in [125] describe a distribution policy based on Socket Cloning that uses a L4/2 front-end, named *Cyclone*. They also select a set of most frequently requested files to be served, in this case, by a set of servers, stored in a hash table in each of the servers. The replication of the files is done progressively as the reference count increases. Hence, if a file's reference count reaches a certain level, the file ends up replicated in all node's caches. Faour and Mansour propose a content-aware distribution policy named *Weblins* in [54] and implement their proposal in Gobelins OS. A cooperative caching mechanism is used to replicate the most requested files in all the nodes of the cluster. They distribute the requests from a layer-4 switch to the servers by the RR algorithm. When a request needs to be transferred to another Web server, it is done by using the TCP Hand-off protocol. Based on WARD policy, Chiang *et al.* implement two distribution policies (*Content-based Workload-Aware Request Distribution with Core Replication (CWARD/CR)* and *Content-based Workload-Aware Request Distribution with Frequency-based Replication (CWARD/FR)*) in a cluster named LVSCAD/FC (LVS with Content-Aware Dispatching and File Caching) in [40]. All the nodes prefetch the set of the most frequently accessed files and the less frequently accessed files are partitioned among the nodes in the CWARD/CR policy,. While in the CWARD/FR policy, the replication of the files in the nodes is proportional to their access frequencies. In both cases, the requests are assigned to servers based on the RR policy among the servers that have the requested file replicated. Results show that CWARD/FR outperforms CWARD/CR.

Persistent connections are not considered in some of the previous proposals [107, 37, 110] because they mean an increase in cost when distributing the load based on locality, as the connection has to be migrated from server to server. The overhead effort of the TCP connections portability among Web servers is studied by Carrera and Bianchini in [29]. They propose a solution named *PRESS* that has two modes of operation. It starts serving requests with a content-blind distribution policy until the cache miss rate reaches a certain threshold, then the policy switches to a locality-aware mechanism as the TCP Hand-off cost is now justified. TCP Hand-off is used when a request has to be served by another server in the cluster. The authors conclude that portability should be promoted in fast-communication clusters as it has a low cost in terms of performance, but efficiency should be promoted in slow-communication cluster as in this case portability is very expensive. Tang and Chanson investigate how request- and session-grained allocations affect caching performance under persistent connections in [131]. They compare two algorithms: *Balanced Content-Based (BCB)* and *Session Affinity-Aware (SAA)*. The authors conclude that the locality-aware performance benefits of request-grained content-aware distribution policies are not easily extended to session-grained allocations.

Ciardo *et al.* in [42] develop a distribution policy named *EQUILOAD* based on the sizes of the requested documents. They partition the sizes of the working set into some subsets (the same as number of servers in the cluster). EQUILOAD is modified in [114] in order to avoid the need of an a priori knowledge of the working set size distribution. The modified solution, named ADAPTLOAD, dynamically sets empirically-based fuzzy boundaries to the intervals of the response sizes. Despite EQUILOAD and ADAPTLOAD not being designed as locality-aware mechanisms, they also obtain the benefits of caching as they always direct requests for the same document to the same server. EQUILOAD and ADAPTLOAD only consider static Web traffic. In a later work [147], ADAPTLOAD is tested with a workload that also includes dynamically generated Web content. As it not possible to know the size of the dynamic content requested, the authors distribute dynamic requests based on the Join Shortest Queue (JSQ) policy. We have included this version as ADAPTLOAD v2 in Table 2.3.

Also considering dynamic traffic, *Harvard Array of Clustered Computers (HACC)* is a request distribution developed by Zhang *et al.* in [150] that dynamically divides the entire working set among the number of nodes in the cluster. The load balancing scheme of this proposal is based on a two-ways architecture that does not take into account persistent connections.

In a recent work [95], Multiple TCP Rebuilding is implemented in LVS platform and named as LVS-CAD. As LVS cannot perform content-aware distribution, a fast TCP module handshaking has been introduced in the IP-layer of the front-end in order to establish the connection with the client. Liu *et al.*, in this work, describe three different distribution policies: *Content-Aware Weighted Least Load (CAWLL)*, *Extended Locality-Aware Request Distribution with Replication Policy (xLARD/R)* and *Content-Aware Hybrid Request Distribution (CAHRD)*. CAWLL is not a locality-aware policy, it selects the least loaded back-end server. xLARD/R is an extension of LARD that considers some additional costs in order to estimate the load of the back-end nodes when taking load balancing decisions. Finally, CAHRD mixes both CAWLL and xLARD/R. It switches to CAWLL when a dynamic request arrives, and to xLARD/R when the request is static. Results show that the locality-awareness is more suitable for static than dynamic requests as CAWLL, and hence, CAHRD, perform better than xLARD/R when more than 30% of the workload corresponds to dynamic Web content.

Most of the works referenced in this subsection do not take into account dynamic Web content [107, 12, 37, 38, 110, 29, 42, 114, 131, 125, 54, 40], as shown in Table 2.3. This is due to the unpredictability of service times of generating dynamic Web pages. In the next subsection, dynamic Web traffic is considered in the design of most of the proposals.

Non locality-aware solutions

From the year 2001, more non locality-aware solutions appear in literature. They do not consider cache hits in the performance evaluation of the Web system. Therefore, other factors are included in these policies as, for instance, the evaluation of the resource utilisation by the different types of requests [31, 143, 123], the introduction of QoS in the service provided [122, 91, 123, 106], or the inclusion of an admission control in the load balancing solution [123].

Considering only the content of the incoming request, Casalicchio and Colajanni propose a policy named *Client-Aware Policy (CAP)* in [31] that obtains good performance results when serving dynamic and secure Web content. As the requests are classified depending on the expected impact they will have on the server resources, CAP takes distributing decisions depending on the service type required by them. The state of the servers is not considered in this solution. All the servers of the cluster provide all the service types considered.

Considering the content of incoming requests and obtaining some metrics from the Web switch that permit to estimate the load of the Web servers, Yao *et al.* in [143] propose *Message-Aware Adaptive (MAA).* It is a scheduling policy that is developed to be executed in the Application Oriented Networking (AON) product of Cisco Systems [1]. This policy distinguishes among types of messages and balances the load based on their resource consumption. Hence, when a request arrives to the system, an estimation of the completion time is calculated and the server that obtains the earliest is chosen to serve the request.

There are other proposals that do include monitored information obtained from the Web servers of the cluster to take distribution decisions. This is the case of *Fuzzy Adaptive Request Distribution (FARD)* [24]. This work of Borzemski and Zatwarnicki describes a content-aware load balancing mechanism that estimates the response time of each request in every server of the cluster using a fuzzy estimation mechanism. This estimation is based on some metrics obtained from the server, such as the CPU, the disk and the communication link load. FARD selects the server with the lowest estimated response time in order to attend the request. The authors compare their proposal to LARD and WRR in a prototype, concluding that FARD improves their performance, specially in heterogeneous environments. Another work that also obtains performance information by monitoring the applications running on the nodes is *Adaptive Load Balancing Mechanism (ALBM)* [41]. Choi describes in this paper a load balancing mechanism that provides scalable services in a multi-cluster system. The front-end of the system is a layer-4 Web switch that balances the load by using DR and NAT, and then the content-aware distribution is done from the nodes of the cluster. ALBM is compared to LVS using RR, LC and WLC scheduling algorithms.

When taking decisions based on information obtained from the servers it is very important that the information is updated. Some authors [48, 118, 123] express their concern about the accuracy of the information obtained as it might not be very realistic in the moment the decision is taken. Worried about the problem of load balancing with stale information, Satake and Inai propose a scheduling method named *Non Probabilistic Server Selection Method (NPSSM)* that obtains the information periodically from the Web servers of the cluster in [118]. Based on this information, the authors try to compensate the differences in load among the nodes and once the load is balanced, they apply the RR policy for the next requests. By simulation, the authors conclude that their method is scalable with the number of servers and that it is immune to the impact of load information acquisition intervals.

QoS requirements are included in some other proposals that also obtain monitored information from the Web server [122, 91]. Considering Stochastic High-Level Petri Net (SHLPN) modelling, Shan *et al.* develop a QoS-aware load balancing strategy named *Extended Fewest Server Processes First (E-FSPF)* in [122]. They combine a process scheduling policy in the Web servers, that considers the priority of the requests, with a load balancing mechanism in the front-end of the system that sends the request to the Web server with the fewest number of processes with higher or equal priorities. Li *et al.* describe in [91] a Web distribution system named *Gage* that balances the load among a set of Web servers and supports QoS. They use a front-end that splices the TCP connection with the chosen server in a one-way architecture. The resource consumption is considered as the QoS metric and the SLA is guaranteed by the allocation of multiple system resources. The requests are scheduled following a WRR algorithm.

When including QoS requirements in a load balancing mechanism, often admission control policies are also introduced in order to avoid a sudden collapse of the servers due to an increase in the demand. The content-aware load balancing algorithm with QoS-aware admission control proposed by Shafirian *et al.* is named IQRD [123]. The authors monitor the Web servers in order to dynamically compute the remaining capacity of the Web system. The requests are classified depending on the resources they consume in the server nodes and are assigned to the nodes based on their load status. The authors compare their strategy to CAP and WRR and conclude that, despite IQRD introducing more overhead, it improves the performance of the system in terms of response time and throughput.

Dynamic traffic is specially considered in [145, 106]. Zhang *et al.* in [145] propose an enhancement to the ADAPTLOAD policy. As the ADAPTLOAD policy described in [147] did not include an original treatment for dynamic requests, the authors study now how the autocorrelation in the arrival process affects the performance of the load balancing policies and propose a load unbalancing mechanism that tries to reduce the autocorrelation of the requested file sizes. They named their policy as *D_EQAL*. In this case, the aim for being locality aware is less clear than in ADAPTLOAD, this is the reason to include D_EQAL as a non locality-aware solution. Ok and Park in [106] describe an algorithm that focuses on dynamically generated Web content (*Around k-Bounded*). As it is not possible to know the load that the execution of the scripts involves in the Web server in advance, they propose an algorithm running in a layer-4 Web switch that monitors the load of the servers. If the servers are attending the same number of requests then the Web switch balances the requests following a RR fashion. The switch sends the next request to the least loaded server in the cluster when the load values of the servers are not equal. Persistent connections when using HTTP/1.1 are considered, hence in case the target server for a request is other than the one that is connected to the client, the TCP Hand-off protocol is used to migrate the connection to the appropriate server. The authors show the benefits of their proposal in terms of scalability and QoS guarantees.

2.6 Summary

This chapter sums up the load balancing mechanisms that have been developed and classifies them by differentiating the OSI protocol stack layer the load balancing is based on. Content-blind load balancing mechanisms have been widely developed in literature and have also been considered to be included in commercial products (some of which have already been withdrawn from the market). As the content-aware load balancing solutions become more popular (mainly because they introduce much more differentiation in the workload and hence, a more accurate distribution of the requests), more effort is dedicated to avoid their drawbacks. Some of these drawbacks are related to the possibility that the layer-7 Web switch becomes the bottleneck of the system and also the request granularity difficulties when using HTTP/1.1 protocol. These problems are solved by using content-blind Web switches and transferring the distribution task to the Web server nodes.

Locality-aware policies have been extensively investigated during the first years of the 2000 decade, with the aim of exploiting the cache performance benefits in the Web servers. Most of these algorithms only consider static content in the Web pages. The fact of considering dynamic Web pages in the workload complicates the load balancing as the service times of the scripts that generate the dynamic content are not easily predictable. Also the cost of generating a dynamic Web page is more expensive in terms of performance than the cost of serving static Web pages and it is difficult to measure or predict. This is still one open problem that needs more research.

The "freshness" of the information obtained from the Web servers that the algorithm uses to take the load balancing decisions is also a problem that has to be more researched, as most of the proposals described here do not consider the possibility that the load information is stale. We are also referring to this point in next chapters.

Table 2.3: Content-aware Request Distribution Policy characteristics: (1) Year; (2) Locality-awareness (Y=yes; N=no); (3) Dynamic content served; (4) Request granularity with HTTP/1.1; (5) One-way architecture; (6) Web Switch layer; (7) Load balancing mechanism

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
LARD [107]	98	Y	N	N	Y	7	TCP Hand-off
extLARD [12]	99	Y	N	Y	Y	7	TCP Hand-off
HACC [150]	99	Y	Y	N	N	7	TCP Splicing
FLEX [37]	01	Y	N	N	Y	3	DNS load balancing
WARD [38]	01	Y	N	Y	Y	4	RR-DNS,Multiple TCP Hand-off
TAP^2 [110]	01	Y	N	N	N	7	TCP Connection Binding
PRESS [29]	01	Y	N	Y	Y	4	TCP Hand-off
CAP [31]	01	N	Y	N	Y	7	TCP Hand-off
EQUILOAD [42]	01	Y	N	N	N	?	Not specified
ADAPTLOAD [114]	01	Y	N	N	N	?	Not specified
E-FSPF [122]	02	N	N	N	N	7	TCP Splicing
FARD [24]	03	N	Y	N	N	7	TCP Connection Binding
Gage [91]	03	N	N	N	Y	7	TCP Splicing
BCB and SAA [131]	03	Y	N	Y	Y	7	TCP Hand-off
Cyclone [125]	04	Y	N	Y	Y	4	Socket Cloning
ALBM [41]	04	N	Y	N	N	4	DR and NAT
ADAPTLOAD v2 [147]	05	Y	Y	N	Y	7	Not specified
D_EQAL [145]	06	N	N	N	?	?	Not specified
Weblins [54]	06	Y	N	N	Y	4	TCP Hand-off
around k-Bounded [106]	06	N	Y	Y	Y	4	TCP Hand-off
NPSSM [118]	06	N	N	N	Y	7	Not specified
CAWLL [95]	07	N	Y	Y	Y	7	Multiple TCP Rebuilding
xLARD/R [95]	07	Y	N	Y	Y	7	Multiple TCP Rebuilding
CAHRD [95]	07	Y	Y	Y	Y	7	Multiple TCP Rebuilding
CWARD/CR [40]	08	Y	N	Y	Y	7	Multiple TCP Rebuilding
CWARD/FR [40]	08	Y	N	Y	Y	7	Multiple TCP Rebuilding
MAA [143]	08	N	Y	N	Y	7	Not specified
IQRD [123]	08	N	Y	N	N	7	Not specified

Chapter 3

Burstiness and Admission Control in a Web system

This chapter briefly describes Internet traffic characteristics in order to establish the context to review burstiness detection and monitoring research in literature. The most significant admission control mechanisms that have been recently proposed are also described. Some parts of this chapter have been previously published in [67].

3.1 Introduction

The Internet community is continuously growing and the need of performance studies in this field is essential in order to obtain better response times for the Web users. The performance of a Web server can be seriously affected when there is a sudden increase in the Web pages demand. This leads to a necessary study of Internet traffic characteristics in order to define the mechanisms that will avoid a congestion in the Web server when it cannot attend all the incoming requests. In order to maintain the performance of a Web system, we have considered two possibilities.

On one hand, a possible prevention is to monitor the traffic arrival related attributes at Web servers and set some policies that will avoid the overload. Hence, a review of the works that analyse the characteristics of Web traffic is included in this chapter. We are more specifically interested in the works that deal with the burstiness of the Web traffic.

On the other hand, it is necessary also to monitor some performance metrics at the Web system's resources continuously to ensure that the service is guaranteed. Moreover, and in case QoS is considered, this monitoring leads to scheduling and admission control decisions that permit user's requests with high SLA to be attended faster and more accurately than others with a lower SLA and also to reject low priority requests when the servers are overloaded.

This chapter is organised in the next different sections:

- The characteristics that are observed in Internet traffic and that may affect Web servers, are briefly detailed in Section 3.2.
- Related studies on burstiness analysis in Internet traffic are considered in Section 3.3.
- Section 3.4 describes related works on admission control and includes a table that sums up the characteristics of these proposals we are most interested in.
- Finally, we include the summary of the chapter.

3.2 How Internet traffic characteristics affect Web servers

The fact that Internet traffic flows exhibit heavy-tailed probability distributions has been widely discussed in the Internet literature [11, 100]. As Web traffic inter-arrival times normally follow a heavy-tailed distribution, maintaining a good performance of the Web system is normally more difficult than if this distribution were easily predictable. Hence, it is possible that in a few seconds a Web server that is not overloaded receives an increase in the number of connections, which produces a situation of congestion [16, 146]. This occurs when the server reaches the connection number limit it can handle. Even, without reaching this limit, if the client's request for a connection is accepted, the response time for that request may experience a long delay because of the long queue of requests waiting to be served by the Web system [146]. The problem would be substantially simplified in the case the Web system were receiving arrivals that follow a constant distribution. Unfortunately this is not applicable on Web traffic. Moreover, Internet traffic's behaviour is based on bursty patterns, then a monitor and control of burstiness in the arrival rate is essential to ensure a good response from the system.

Internet service providers often offer different QoS levels to supply different priorities to different users. When the congestion situation is severe, admission policies are normally applied. This leads to the challenge of satisfying the performance requirements for different types of requests at all times.

In the next sections we review the studies that appear in literature related to burstiness detection and admission control in a Web system.

3.3 Burstiness in Internet

In this section, burstiness modelling and detection related research is introduced. We have classified the related work depending on the context the burstiness is analysed. Hence, we have distinguished among burstiness detected in network traffic, in TCP protocol, in databases and in service times.

3.3.1 Burstiness detection based on network traffic

The pioneers in modelling burstiness in a network are Wang *et al.* [137]. They analyse the relation between jitter and burstiness in real-time communications. In this paper, the burstiness detection mechanism is defined for individual packets. The authors define the burstiness of the mth packet as a measure of time that expresses the distance between the actual arrival time and the right edge of the mth packet arrival interval because they consider the servers usually process packets one by one at a constant rate. An implementation of this mechanism has been defined in our simulation scenario. More details and results are described in Section 4.3.

Burstiness detection based on the traffic rate and independent of individual packets is defined in other papers [101, 20, 134, 92, 75]. Menascé and Almeida [101] are the first to introduce a burstiness factor. They define it as the fraction of time during the time slot arrival rate that exceeds the average arrival. In this case, burstiness monitoring is carried out following fixed time slot scheduling. In Section 4.3, we describe in detail how this burstiness factor is defined and works and introduce some modifications to it. Baryshnikov *et al.* [20] study how traffic predictions can be very useful to reduce latencies and performance degradation in Web servers. They use linear extrapolation as a prediction technique and state that this technique for burstiness detection is not a good predictor. In general, however, they conclude that even simple prediction algorithms have a significant prediction power. We also consider their mechanism in the burstiness factors we define in Section 4.3. In [134], van de Meent *et al.* detect burstiness through the average traffic rate and the peak rate each second. They define a non-linear relation between these two variables to model the variation in the traffic rate that shows burstiness. Li *et al.* in [92] detect burstiness in their model by defining thresholds. If the arrival rate exceeds the thresholds in a number of successive slots, then sustained burstiness occurs. In [75], Gusella characterises burstiness of packet arrival process with indexes of dispersion for inter-arrival times and packet counts, that are based on their variance.

Other papers deal with the characteristics of Internet traffic by focusing on the burstiness implicit to it. Sarvotham *et al.* [117] define an alpha/beta traffic model, considering the traffic bursts as the alfa-traffic and analyse why the bursts occur in network traffic. Lan *et al.* in [90] define a burst as a train of packets with a lower inter-arrival time than a threshold and study the correlations between size, rate and burstiness.

3.3.2 Burstiness detection based on TCP protocol

Burstiness has also been modelled for TCP traffic in other studies such as [58, 138]. In [58], the authors detect bursts of TCP acknowledgement packet transmissions in order to increase the size of the congestion window. A burstiness model is defined in [138] that assigns a burstiness value to each TCP packet based on the Round-Trip Time (RTT) in order to control the actual sending rate.

3.3.3 Burstiness detection in databases

With regards to databases, Vlachos *et al.* [135] detect short-term and long-term bursts in on-line search queries by comparing the moving average (of 7 days and 30 days for short-term and long-term bursts, respectively) of user demands with its mean value.

3.3.4 Burstiness detection in service times

A new methodology for capacity planning under bursty workload conditions is defined by Mi *et al.* in [102]. They introduce a index of dispersion based on the one proposed in [75], that detects variability and burstiness. They test their index in a multi-tier e-commerce site based on TCP-W specifications.

3.4 Admission control policies

The problem of the admission control in a Web server has been widely addressed in literature. The most significant Web admission control related algorithms are introduced in this section. We have included in Table 3.1 a summary of the characteristics we consider the most important of the admission control proposals cited in this section, that are organised in chronological order as it is indicated in the row (1) of the table.

3.4.1 QoS-aware scheduling policies

There are several pioneering QoS-aware scheduling policies that should be mentioned in this subsection, despite no explicit admission control mechanism is introduced. We are referring to the work of Almeida *et al.* [7], Pandey *et al.* [108] and Eggert and Heidemann [51]. They develop priority-based scheduling methods that permit to differentiate between several types of traffic in the Web server and require modifications on the Web server [108, 51] or both the Web server and the OS kernel [7]. In [7], the authors develop a strategy to improve the response time of the most priority requests by controlling the number of processes in the Web server. In [108], an implementation of a distributed HTTP server that sets resource usage constraints to the requests based on their priority is described. In [51], three different application-level mechanisms for a Web server are considered that provide different levels of Web service. QoS-awareness of the related work is indicated in the row (2) of Table 3.1.

3.4.2 Classic Control Theory-based admission control policies

Feedback control theory has been used by some authors in admission control algorithms [6, 136, 84]. It is included in the row (3) of Table 3.1. An admission control strategy that includes QoS differentiation is detailed by Abdelzaher *et al.* in [6]. They propose a mechanism based on the classical control theory to guarantee the performance of requests by establishing a deadline for each request. A CPU monitor is used to maintain the utilisation at a threshold by using a classical linear feedback control problem formulation. The admission control decision is taken for a subset of requests based on these server utilisation measures. Voigt and Gunningberg in [136] propose a request-based admission control that includes service differentiation. It is based on feedback control loops that determine the maximum rate of accepted requests. Based on the monitored performance metrics the algorithm decides to accept or drop a SYN request. Kihl *et al.* [84] consider control theory to develop a non QoS-aware admission control mechanism for new requests that monitors server's CPU utilisation at fixed time intervals.

3.4.3 E-commerce-based admission control policies

Other works describe proposals that consider session-based workload and that are defined for e-commerce environments [82, 15, 39, 85, 34, 52, 19, 111]. It is indicated in the row (4) of Table 3.1. Kanodia and Knightly in [82] develop a multi-class session admission control based on latency targets. A class-based admission control decision is taken when a new session request arrives to the system. The decision considers whether the request class target delay is guaranteed, or not, and the effect its acceptance will have on the latency of the rest of the classes. Considering also session-based workload, Aweya *et al.* [15] propose a load balancing scheme in a cluster of Web servers that includes an admission control algorithm based on the CPU utilisation metric, that is retrieved from the Web servers at fixed intervals. The acceptance rate of client requests is adaptively set based on the CPU performance measures. Cherkasova and Phaal in [39], consider the rejection of sessions when the Web server is overloaded in order to avoid the consumption of the server resources by a user session that may be interrupted. The metric used to monitor the Web system performance is the CPU utilisation, which is measured during predefined time intervals and used to compute a predicted utilisation. In case the predicted utilisation gets above a threshold, then all the new sessions that arrive for the next interval are rejected, but the service of already accepted sessions continues. Sessions are admitted again when the predicted utilisation goes below the threshold. In [85], Kihl and Widell monitor the processing delay of each request and, based on this information, the admission control algorithm considers the admission, or not, of a new session or request on commercial QoS-aware Web sites. They model the arrivals to the Web system according to a Poisson process. Chen and Mohapatra propose a capacity planning to avoid overload in Web servers and design a scheduling scheme based also on the session-level traffic model in [34]. They design a Dynamic Weighted Fair Sharing (DWFS) scheduling algorithm to control overloads in Web servers that is based on the probability of session completion. Elnikety *et al.* in [52] develop an admission control and request scheduling for e-commerce environments also based on sessions. They consider the TPC-W model to classify the workload in transactions. The admission control algorithm estimates the load that a particular job will generate and the capacity of the system. The job is delayed in a Shortest-Job First (SJF) queue in case of overload. The load of the system is measured each second, but the admission control is executed each time a servlet requests a database connection. One of the benefits of this proposal is that it does not require any modification in the operative system nor the Web software (Web server, application server or database). Bartolini *et al.* describe in [19] a policy that switches between two modes depending on the arrival rate detected at the system. When the system is not overloaded, their approach takes admission control decisions at fixed intervals of time. In case the arrival rate exceeds a limit, then the admission control decisions are taken each time a new session arrives to the system. They limit the 95th percentile of the response time of each type of requests. Another proposal for an e-commerce environment is described by Poggi *et al.* in [111]. They describe a system that learns the user's behaviour and obtains the predicted probability of purchase that a new session has. The prediction is updated periodically (daily, weekly, etc.). They obtain this prediction off-line based on a log of an e-commerce site. During an overload situation, the new session is accepted or denied based on this prediction.

3.4.4 Web cluster-based admission control policies

Some proposals consider a Web system composed of more than one Web server, that normally is referred as Web cluster or Web cluster-based network servers [108, 82, 15, 85, 19, 111, 22, 152, 133, 123]. The row (5) of Table 3.1 details this characteristic. As some of them have already been commented on above, we describe the rest here. Bhatti and Friedrich [22] consider a tiered architecture that includes an admission control mechanism that is based on two fixed thresholds: one that counts the total number of requests in the system and another that counts the number of the most priority requests. The rejection of less priority requests starts when one of the thresholds is reached. In this case, the performance metric that is monitored is the number of requests queued in the system. Zhou and Yang [152] describe an early request termination policy for the long requests that may affect the performance of a Web server. They monitor the load of the server and the running time of the requests and, based on them, set an adaptive termination threshold for each request. Their proposal includes a resource accounting module that ensures that all the allocated resources are correctly deallocated when a request is terminated. Sharifian *et al.*

in [123] describe a load balancing algorithm that includes an admission control policy for a cluster of Web servers. They differentiate the requests depending on their service times and estimate the load of the Web server using a queueing model that is based on performance metrics that are obtained periodically. Rejection starts when there is no remaining capacity for a specific class of requests. Urgaonkar and Shenoy in [133] propose a very interesting low-overhead admission control algorithm for handling extreme overloads. In this work, the authors introduce some dynamically adjusted class-based delays to invoke the admission control algorithm and vary these delays with the workload changes when the system is not overloaded. They switch the admission control policy to a periodical threshold-based strategy when the load increases.

3.4.5 Other types of admission control policies

There are also other works that should be mentioned. Chen *et al.* in [35] describe an admission control algorithm that includes differentiated services. They try to guarantee a maximum response time by adjusting periodically the resource allocation of the system on the basis of a service time prediction for each request class. They monitor the CPU cycles in a previous experiment in order to know the service time of the requests. The admission control is done following a per-request basis. Cheng *et al.* in [36] develop a threshold-based admission control algorithm with different priorities that searches a sub-optimal solution to adjust the threshold values when the system workload changes. They compare their solution with an optimal solution obtained by a Petri net. We also include the work of Welsh and Culler [139] who propose an approach that split the Web services in several stages, each of them with a specific admission controller. They use the 90th-percentile of the response time as the performance metric used to drive overload control. The response time is measured for each request that passes through a stage, but the admission control decision is taken based on the 90th-percentile response time over some interval. Andersson *et al.* in [8] describe an admission control algorithm that is based on static thresholds for the response time of different classes of requests. The performance metrics that are monitored by the algorithm are the CPU utilisation, the arrival rate and the response time of the classes of requests. They apply a linear optimisation algorithm to maximise the total revenue for the Web site during overload. Schroeder and Harchol-Balter, in [119], avoid overload by executing an unfair connection schedule instead of an admission control policy that would lead to the deny of the service to some requests. They give priority to requests that demand small static content files by using the Shortest-Remaining-Processing-Time-First (SRPT) scheduling algorithm. Their proposal requires modifications at the kernel level of the OS.

Table 3.1 sums up some of the characteristics of these previous works, as it is difficult to classify most of them in less than one category. The columns are the references of the previous works in chronological order. Rows (1)-(5) have already been commented on above. The row (6) indicates the performance metric that is monitored by the algorithm and the row (7) considers the invocation frequency of the admission control algorithm, that normally is the same as the monitoring frequency of the performance metric that is used as the input of the algorithm. This performance metric is needed to take admission control decisions and can be demanded in different ways depending on the proposal. Some authors define a fixed time interval to obtain monitored performance values and invoke the admission control algorithm [6, 15, 39, 139, 123, 84, 111], while others check the performance metric selected and then execute the admission control algorithm when a determined event has occurred, i.e. each time a new request or session arrives to the Web system [7, 108, 22, 51, 82, 85, 136, 35, 34, 36, 52, 8, 119, 152]. Two recent works introduce a dynamic variation of the invocation times of the admission control algorithm that depend on the workload [133, 19] with the goal of overhead reduction. In both cases, there is a switch in the admission control policy depending on the overload of the Web system.

3.5 Summary

This chapter reviews the literature of two technical aspects: burstiness in Internet and admission control policies. Despite at first glance they are not very related, it is a fact that the burstiness in arrival rate affects to the Web servers' performance and might cause a congestion when it is not treated adequately.

As we have seen in Section 3.3 the related research on burstiness varies depending on the context it is studied. We are most interested in burstiness detection on network traffic. As indicated above, we have included some ideas from some papers [137, 101, 20] in our burstiness definition, which is detailed in the next chapter. We have also used the non-linear relation defined in [134] to analyse some of the results obtained.

Referring to the admission control proposals, it can be observed in Table 3.1 that not many of them consider the arrival rate, and then the burstiness, among the performance metrics the admission control mechanism is based on [15, 136, 34, 8, 123, 133]. Hence changes in the arrival rate are not detected. Furthermore, in some proposals, the invocation frequency of the admission control is periodical with the aim of not introducing an excessive overhead in the system.

Among the proposals that include a non-periodical invocation frequency, only two of them do not consider the arrival of a new session or request as the trigger to execute the algorithm [133, 19]. These two recent works introduce a two different invocation methods, one that acts when the system is not overload and the other that acts when it is. In fact, it catches our attention that while [133] switches to a periodical threshold-based strategy when the system is overloaded, [19] starts taking admission control decisions each time a new session arrives to the system. In both cases, there is a switch in the admission control policy depending on how overloaded the Web system is. We consider these two works a step forwards in the planning of the invocation of an admission control algorithm. It is a subject that needs to be more researched. Our invocation scheduling proposal is described in the next chapters.

Table 3.1: Main characteristics of admission control policies: (1) The year the reference is published; (2) QoS-aware algorithm (Y=yes, N=no); (3) Classic Control Theory-based; (4) Session-aware; (5) Cluster of Web servers considered; (6) Performance metric monitored (1=CPU utilisation, 2=service or response time, 3=number of requests, 4=arrival rate, 5=transaction size, 6=number of processes running in the Web server) and (7) Invocation Frequency (P=Periodical, NP=Non Periodical).

	Reference:	[7]	[108]	[22]	[51]	[82]	[6]	[15]	[39]	[85]	[136]	[35]	[34]	[36]	[139]	[52]	[8]	[119]	[152]	[84]	[123]	[133]	[19]	[111]
(1)	Year	98	98	99	99	00	02	02	02	02	02	03	03	03	03	04	05	06	06	08	08	08	09	09
(2)	QoS-aware	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y
(3)	Feedback Control Theory	N	N	N	N	N	Y	N	N	N	Y	N	N	N	N	N	N	N	N	Y	N	N	N	N
(4)	Session aware	N	N	N	N	Y	N	Y	Y	Y	N	N	Y	N	N	Y	N	N	N	N	N	N	Y	Y
(5)	Web Cluster	N	Y	Y	N	Y	N	Y	N	Y	N	N	N	N	N	N	N	N	Y	N	Y	Y	Y	Y
(6)	Performance metric	2,6	1-3,5	3	6	2	1	1,4	1	2	1,4	2	2,4	3	2	1,2	1-2,4	5	1-3,5	1	1-3	2-4	2,4	3
(7)	Invocation Frequency	NP	NP	NP	NP	NP	P	P	P	NP	NP	NP	NP	NP	P	NP	NP	NP	NP	P	P	NP	NP	P

Part II

Contribution

Chapter 4

Analysis of Burstiness Monitoring and Detection

Due to the heavy tailed pattern of Internet traffic, it is crucial to monitor the incoming arrival rate in a Web system to preserve its performance. In this chapter, we focus on the arrival rate processing mechanism as part of the design of an adaptive admission control and load balancing Web algorithm. The arrival rate is one of the most important metrics to be monitored in a Web site to avoid the possible congestion of Web servers. Six methods are analysed to detect the burstiness of incoming traffic in a Web system. We define six burstiness factors to be individually included in an adaptive admission control and load balancing algorithm, which also needs to monitor some Web servers' parameters continuously, such as the arrival rate at the server or its CPU utilization in order to avoid an unexpected overload situation.

We also define adaptive time slot scheduling based on the burstiness factor, which reduces considerably the overhead of the monitoring process by increasing the monitoring frequency when bursty traffic arrives at the system and by decreasing the frequency when no bursts are detected in the arrival rate. Simulation results of the behaviour of the six burstiness factors and the adaptive time slot scheduling when sudden changes are detected in the arrival rate are presented and discussed. We have considered a scenario made up of a locally distributed cluster-based Web information system for simulations. An early study of burstiness monitoring and detection was included in [66]. Most of the work in this chapter has been previously published in [67].

4.1 Introduction

Our main concern in the design of an admission control and load balancing Web system is how to monitor some Web servers' parameters in a very adaptive way in order to reduce the algorithm overhead. Some of the Web servers' parameters likely to be monitored are the arrival rate, the CPU/disk utilisation, I/O performance, etc. The performance of the nodes that compound the Web system have to be monitored continuously in order to know their status and make the appropriate decisions in case of overload to avoid a possible congestion situation. This can be done in several ways: *(i)* each time a request arrives at the front-end of the Web system; *(ii)* at fixed times by using static time slot scheduling; or *(iii)* at non-fixed times by using dynamic time slot scheduling. The overhead introduced by option *(i)* is the biggest because each time a request arrives at the Web system, Web node parameters are monitored. While option *(ii)* introduces a constant overhead, option *(iii)* monitors the system at non-fixed intervals, hence, its overhead will depend on the frequency of those intervals. The drawback of defining monitoring in a constant duration interval schedule (option *(ii)*) is the choice of monitoring time interval. It is very difficult to set a duration interval that fits with all possible Internet arrival rates at the Web system due to its heavy tailed pattern.

We propose using adaptive time slot scheduling (option *(iii)*) where the frequency of monitoring depends on a burstiness factor that will increase its value when bursty arrivals reach the system, and decrease it, if no burstiness is detected. The adaptive time interval we define depends directly on this burstiness factor. Therefore, the monitoring task's overhead is related to the burstiness of the arrivals in the Web system and time slot scheduling is completely adaptive to the burstiness detected in the arrival rate that reaches the system.

We have included six burstiness factors in a content-aware load balancing model designed with OPNET Modeler [3] to compare the effect of including different burstiness factors in the Web system performance. The admission control and load balancing algorithm used is fully described in the next chapter.

The following sections of this chapter are organised as follows:

- Section 4.2 details the definition of monitoring slots.
- The burstiness factors we have considered for our experiments are detailed in Section 4.3.

- The adaptive time slot scheduling mechanism is described in Section 4.4.
- Section 4.5 details the simulation scenario and shows the results obtained.
- Finally we include the summary of the chapter.

4.2 Defining Monitoring Slots

The fact of introducing monitoring to a real system may introduce a relative error in the measurements taken. The most precise measuring of the system parameters could be carried out if monitoring is executed continuously and then all the changes in the system parameters are recorded in a database. The main problem involved is the extremely high overhead that is introduced in the system when many hundreds of requests per second arrive in the Web system and the monitoring process may modify the system's performance. Therefore, the monitoring itself may vary the values of the monitored parameters.

Moreover, obtaining average times of the values measured is a usual technique, even if all the system components are precisely and completely measured. This also leads to an implicit error in the accuracy of the final value of the observed metrics. We consider that this error cannot be avoided because without it, monitoring a system would get bogged down in the extraction and determination of minute details, which may make the overall analysis more difficult.

As previously stated, we propose monitoring the system by using adaptive time slot scheduling. To our knowledge there are no prior studies that set this scheduling to monitor the HTTP arrival rate at a Web system.

Obviously, an implicit error will be introduced. Let us analyse this fact with an example. Suppose we are monitoring the arrival rate of HTTP requests that arrive at the front-end of a Web system. It is evident that the arrival rate of the incoming requests is a random metric. If it is monitored by using fixed intervals, the resulting curve is different to the curve of the arrival rate monitored following adaptive time slot scheduling, which is due to the different values in the x-axis. Figure 4.1 represents both curves and the monitoring intervals scheduled for each one. At the bottom of the plot, the observation times of the arrival rate monitored following a fixed time slot are illustrated. In this case the arrival rate is monitored every 20 seconds. At the top of the plot, the monitored time intervals are represented when following an adaptive time slot. It is clear that on average both arrival

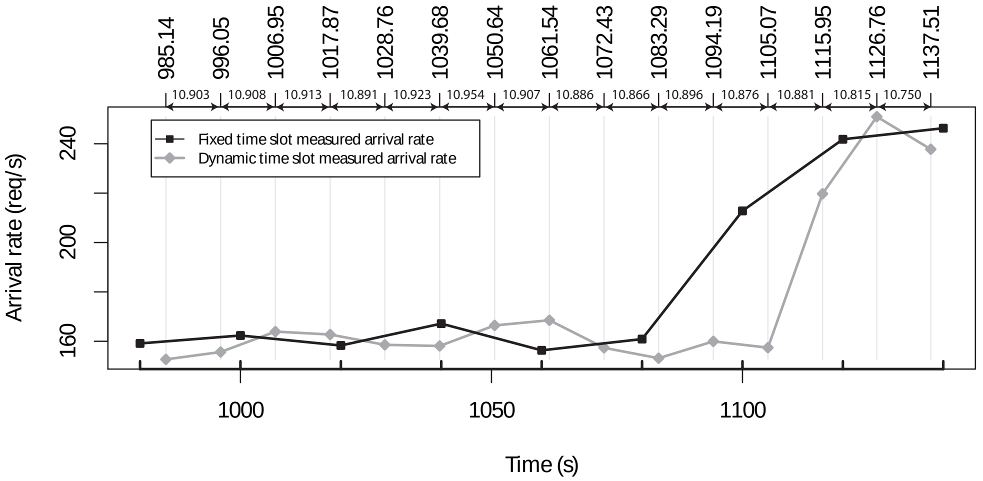


Figure 4.1: Arrival rate monitored following different observation times

rate curves are equivalent but at each moment of the observation period one varies with respect to the other because of the different observation times. In Section 4.4, we explain how we set the duration of the adaptive monitoring time slots.

We are going to define the burstiness factors considered in this chapter in order to study their behaviour, and then we detail how the adaptive time slot scheduling is defined.

4.3 Burstiness Factors

We have considered six different approaches to define burstiness factors in order to compare their behaviour and detect their benefits or drawbacks under the same circumstances. Some of these approaches are based on previous articles we cited in Section 3.3 and the rest are modifications of them, as indicated below.

All the burstiness factor values are defined in [0,1] in order to limit the range of the factor because its value will be used later in the algorithm, for instance, to make some decisions to avoid a congestion situation. Hence, all the burstiness factors defined here can be easily adapted to be used in an admission control algorithm, despite being beyond the scope of this chapter. The factor calculation is made for each time interval or slot defined by adaptive time slot scheduling, that is described in Section 4.4.

We start with the burstiness factor defined in [101] (BF1) and we propose some modifications to it to try to adapt it more accurately to the variations of incoming traffic (BF2, BF3, BF4). Some modifications are also included to introduce linear extrapolation in order to detect the bursty slots (BF5), as Baryshnikov *et al.* suggested in [20]. We have also considered a sixth burstiness factor (BF6) by including the proposal of Wang *et al.* [137]. In this case, the factor is computed for each incoming HTTP request to a Web server.

A description of each burstiness factor is given in the following subsections, and at the same time the reader can observe a representation of them in Figures 4.2a - 4.2j. We have simulated all the burstiness factors proposed in this section in a discrete event simulator, OPNET Modeler [3]. The scenario consists of 5 Web servers that receive requests from 30 clients. The workload is fully described in Section 4.5. We show the behaviour of the factors we propose together with the arrival rate monitored in a Web server plotted on two scales (on the right the burstiness factor scale; on the left, the arrival rate scale). At the bottom of the figures the monitor time intervals of the arrival rate is given. All of the plots included in Figure 4.2 have been obtained from the same simulation scenario that receive exactly the same workload, although some variations in the arrival rate can be observed. These variations are due to the different lengths of the time intervals when we monitor the system, as explained in the previous section, which depends directly on the burstiness factor applied .

4.3.1 BF1: Menascé proposal

The first burstiness factor we use was proposed by Menascé and Almeida in [101]. Its definition requires knowing the mean arrival rate of HTTP transactions for a Web server measured during some time intervals or slots $0, 1, ..., k-1$, denoted as μ_λ. For a slot k and a Web server, $\lambda(k)$ represents its corresponding arrival rate. If $\lambda(k) > \mu_\lambda$, then the slot is considered a *bursty* slot. Given m slots, the burstiness factor is defined as the relation between the cumulative number of slots that satisfy $\lambda(k) > \mu_\lambda$, called k^+, and the current number of slots, k [101]:

$$b_{orig}(k) = \frac{k^+}{k} \tag{4.1}$$

This burstiness factor smooths the arrival rate curve. Figure 4.2a illustrates that it follows the arrival rate but does not accurately represent its quick variations. We consider that the burstiness factor should alert the system as quickly as possible of an increase in the arrival rate, and this factor increases or decreases along with the increasing or decreasing arrival rate trend but very slowly and delayed.

We propose the direct inclusion of the arrival rate value in the burstiness factor in the next proposal, as a way to modify it quantitatively.

4.3.2 BF2: Arrival rate included

The second burstiness factor considered modifies the previous one by including the relative difference of the arrival rate of the two previous slots. Hence, the burstiness factor modification also depends on how much the arrival rate increases or decreases. Its expression is the following:

$$b_{arr\_rate}(k) = \frac{k^{+}}{k} \cdot \left(1 + \frac{\lambda(k) - \lambda(k-1)}{\lambda(k-1)}\right), \qquad 0 < b_{arr\_rate}(k) < 1 \tag{4.2}$$

Notice that $b_{arr\_rate}(k)$ can be greater than 1 in this definition. When this happens we set it to 1 to fulfill the $0 < b_{arr\_rate}(k) < 1$ restriction.

Figure 4.2b shows that, in this case, the burstiness factor also varies with the variations of the arrival rate. Nevertheless, there are some peaks in the arrival rate that are not followed by the factor. In the next proposal, we introduce a penalisation when detecting a consecutive number of *bursty* slots.

4.3.3 BF3: Penalisation included

We also want the burstiness factor to accurately represent the increasing traffic peaks incoming to a Web server. Hence, we consider that a maximum of j consecutive *bursty* slots should cause a proportional increase in the burstiness factor. This is the reason for including a penalisation in the factor that depends on a record of previous *bursty* slots. This penalisation is limited with a record of j slots, being $\alpha = 0.1 \cdot j$, for $j \in 1, .., 10$. We have chosen to have a maximum record of 10 slots to penalise the burstiness factor because if the burstiness detected in the arrival rate is extreme, then the burstiness factor will be doubled every 10 slots. Hence, the maximum value of the burstiness factor can be easily reached.

$$b_{penalis(j)}(k) = \frac{k^{+}}{k} \cdot (1 + \alpha) \, , \; 0 < b_{penalis(j)}(k) < 1 \tag{4.3}$$

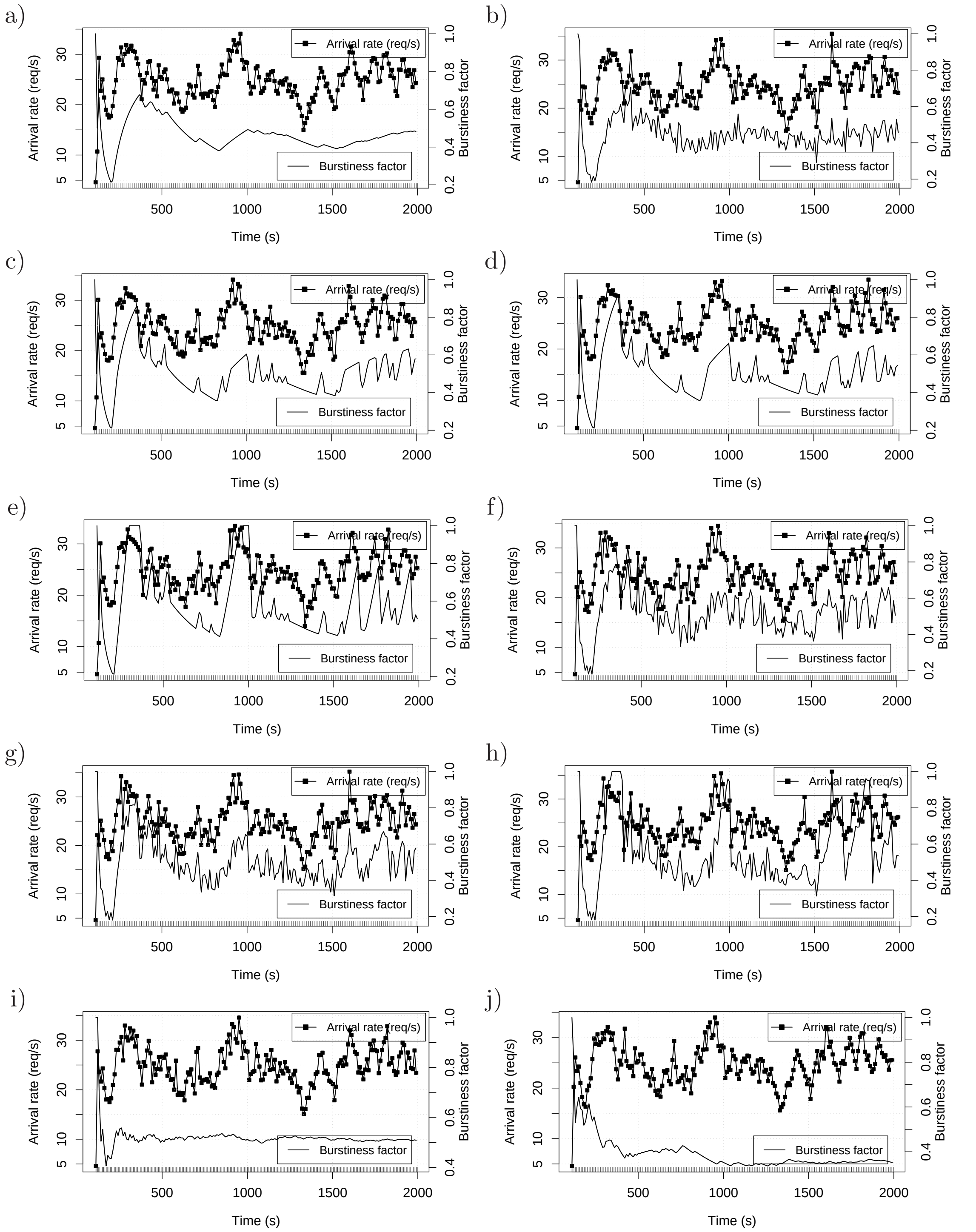


Figure 4.2: Arrival rate and burstiness factors: a) BF1; b) BF2; c) BF3 with $j = 3$; d) BF3 with $j = 4$; e) BF3 with $j = 10$; f) BF4 with $j = 3$; g) BF4 with $j = 4$; h) BF4 with $j = 10$; i) BF5; j) BF6.

We have simulated the scenario with different values of j to compare its behaviour. Figures 4.2c, 4.2d and 4.2e represent the results obtained with this burstiness factor and a record of 3, 4 and 10 slots, respectively. We have omitted the plots corresponding to the rest of the values of j in Figure 4.2 because we consider that the behaviour of this factor is perfectly understood with these three values of j and in this way we avoid the inclusion of more plots. It can be observed that as j increases the burstiness factor penalisation also increases. We need to check if this penalisation (possibly excessive for high values of j) leads to an increase in the system performance or otherwise, decreases its performance because of an overreaction to the arrival rate.

4.3.4 BF4: Arrival rate and penalisation included

This proposal includes the relative arrival rate difference between the last two slots, which will permit the burstiness factor to follow all the arrival rate variations, and the penalisation.

$$b_{arr\_rate\_penalis(j)}(k) = \frac{k^{+}}{k} \cdot \left(1 + \frac{\lambda(k) - \lambda(k-1)}{\lambda(k-1)} + \alpha\right) \tag{4.4}$$

We also limit the value of $b_{arr\_rate\_penalis(j)}$:

$$0 < b_{arr\_rate\_penalis(j)} < 1.$$

The burstiness factor values in the simulation are shown in Figures 4.2f, 4.2g and 4.2h, representing the results obtained with a maximum record of 3, 4 and 10 slots. We can observe that the resulting curves of BF4 are similar to the BF3 curves, but in this case the burstiness factor is also sensitive to changes in the arrival rate.

4.3.5 BF5: Linear extrapolation approach

In this case, we have used linear extrapolation of the arrival rate in order to detect bursty slots instead of the average of the arrival rate, as described by Baryshnikov *et al.* in [20]. We compute the prediction of the arrival rate in the next slot for a Web server as the next expression, $t\,(k)$ being the final time of the slot k:

$$\hat{\lambda}(k) = \lambda(k-2) + \frac{t(k) - t(k-2)}{t(k-1) - t(k-2)} \cdot (\lambda(k-1) - \lambda(k-2)) \tag{4.5}$$

We consider a *bursty* slot when $\lambda(k) > \hat{\lambda}(k)$ and then we apply expression (4.1), considering k^+ as the number of *bursty* slots and k as the current number of slots. This burstiness factor will be represented as $b_{extrap}(k)$.

Figure 4.2i shows the results obtained with this burstiness factor and the resulting curve can be observed as being even smoother than the one obtained from the original Menascé proposal.

4.3.6 BF6: Wang approach

This last proposal, introduced by Wang *et al.* in [137], has been considered in order to analyse its behaviour and compare it with the other proposals. The main difference between this proposal and all the previous ones is that this factor is computed for each incoming HTTP request that reaches a Web server.

Considering $t(1)$ and $t(m)$ as the times when packets 1 and m arrive at the Web server, and $c(t(1), t(m)) = m - 1$ as the number of packets between $t(m)$ and $t(1)$, the expression that Wang *et al.* used to represent the burstiness of each packet is the following [137]:

$$b(m) = t(1) + \frac{c(t(1), t(m))}{\rho_{min}} - t(m) \tag{4.6}$$

where ρ_{min} represents the minimum throughput:

$$\rho_{min} = min\left[\frac{c(t(1),t(2))}{t(2)-t(1)}, \frac{c(t(1),t(3))}{t(3)-t(1)}, \cdots, \frac{c(t(1),t(m))}{t(m)-t(1)}\right]$$

In this case, burstiness is calculated in seconds and represents a measure of time that expresses the distance between the actual arrival time and the right edge of the mth packet arrival interval. As this definition of burstiness is expressed in seconds, we have modified it to be applicable to slots.

Hence, $\mu_b(k)$ being the mean of the burstiness of the packets that arrive in a slot k, and μ_b the mean of the burstiness of all the previous packets, we consider a *bursty* slot when $\mu_b(k) > \mu_b$ and then we apply expression (1), considering k^+ as the number of *bursty* slots and k as the current number of slots. This burstiness factor will be represented as $b_{wang}(k)$.

In Figure 4.2j, it can be observed that the BF6 curve does not accurately follow the arrival rate changes. The BF6 curve decreases in some points of Figure 4.2j when the arrival rate curve increases. We will obtain more results with this burstiness factor in order to know its possible benefits with different workloads despite the fact that its calculation

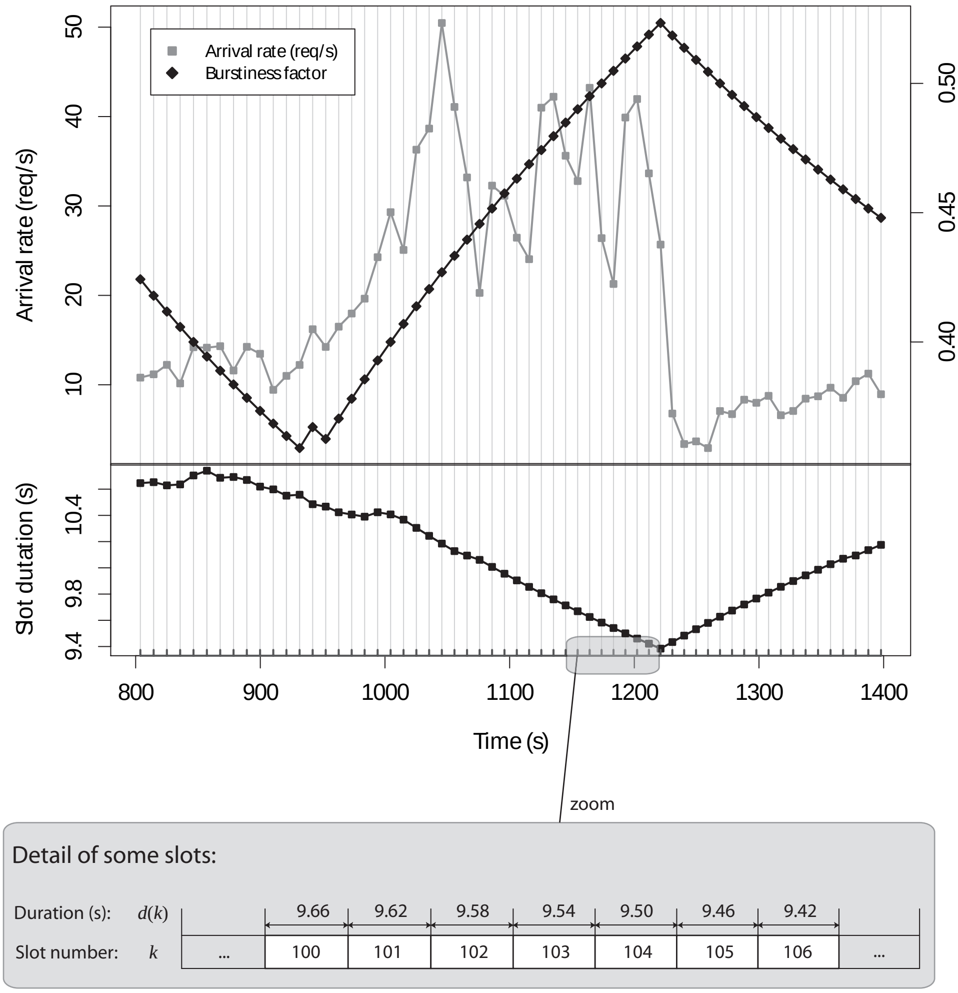


Figure 4.3: Arrival rate monitored following adaptive time slot scheduling and detail of some of the slots using the BF1.

is made for each incoming HTTP request and then it needs a huge computational effort, which leads to a considerable overhead compared to the other proposals.

4.4 Adaptive Time Slot Scheduling

Once we have defined the burstiness factors, let us describe how we use them to set up adaptive time slot scheduling.

We divide the total observation time T of the experiment into several slots of variable duration. While the experiment is simulated, the duration of the slot changes based on the value obtained by the burstiness factor. Hence, the duration of the slot $k + 1$ is dependent on the burstiness of the two previous slots, $b(k)$ and $b(k-1)$, as follows:

$$\begin{aligned} d(k+1) &= \frac{d(k)}{1+b(k)+b(k-1)},\ if\ b(k) \geq b(k-1) \\ d(k+1) &= \frac{d(k)}{1+b(k)-b(k-1)},\ if\ b(k) < b(k-1) \end{aligned} \tag{4.7}$$

Therefore, the number of slots defined during the simulation time is also variable. We can calculate the total number of slots that divide the observation time T during each slot. Considering the duration of the slot $k+1$, the frequency of slots is defined as:

$$e(k+1) = \tfrac{T}{d(k+1)}$$

The burstiness factor will never be 0 because we consider the first slot of the experiment as *bursty* to avoid a division by 0.

As the duration of the following slot is defined by the value of the burstiness factor on the current slot, when a burstiness increase is detected, the following testing time is brought nearer in order to check the incoming arrival rate early enough and then tune again the algorithm parameters. If a decrease in burstiness is perceived, the duration of the following slot is enlarged to reduce the overhead. By controlling the burstiness in the arrival rate, and then the duration of testing slots, a sudden reduction in the future performance of the Web servers may be forecasted.

An example of adaptive time slot scheduling is depicted in Figure 4.3. In the upper part of the figure the arrival rate and the burstiness factor curve are drawn following adaptive time slot scheduling. As the arrival rate increases from time instant 910 seconds, the burstiness factor also increases. We have used BF1 to illustrate burstiness factor behaviour in this case. Below this figure, the slot duration is represented in another scale. It can be observed how the duration of the slots decreases when the arrival rate increases. Some slots have been zoomed in to detail the decrease of their durations.

4.5 Simulation Scenario and Results

We have tested the six burstiness factors described in Section 4.3 in the discrete event simulator OPNET Modeler [3]. The architecture is made up of a set of clients that request Web contents from the Web system as depicted in Figure 4.4. The main advantage of

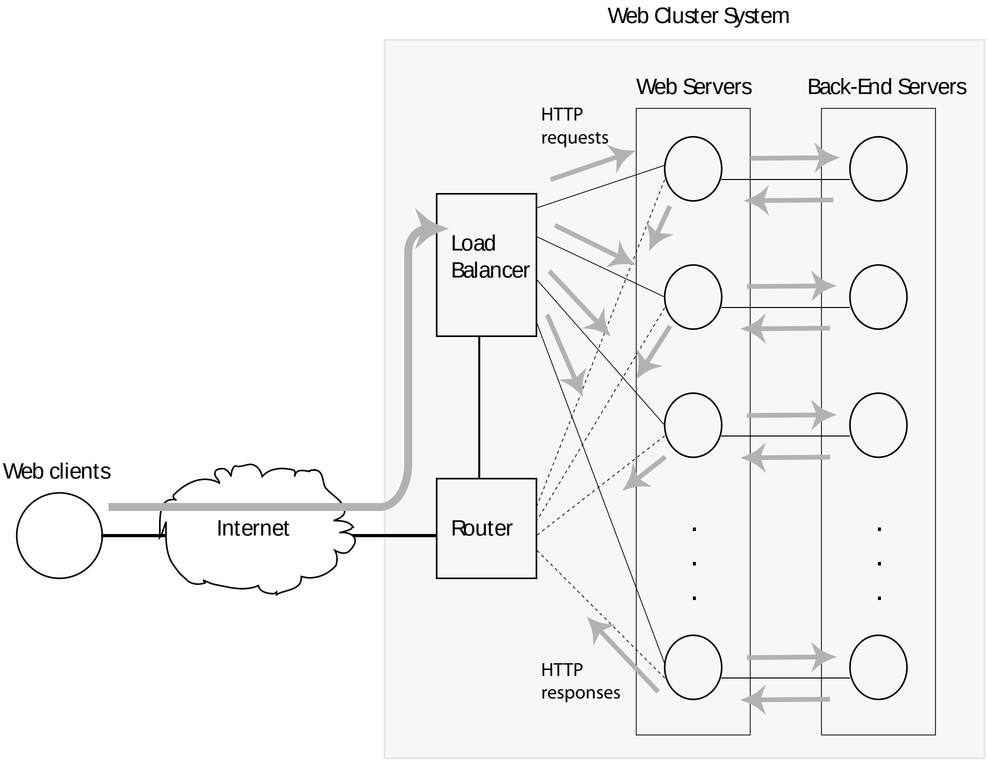


Figure 4.4: The Web architecture is made up of 20 mirrored Web servers and their corresponding database servers. All the Web servers are connected to a load balancer that distributes the load. The model architecture is one-way, which means that the incoming HTTP requests go through the load balancer but their HTTP responses use a different way to prevent the load balancer from becoming the system bottleneck.

defining a burstiness factor is that is can be located in any part of the Web system that receives an arrival rate. We have chosen to detect the burstiness in the Web servers of the Web system [68, 65]. The admission control and load balancing algorithm employed in this scenario is fully described next chapter. The architecture is modeled as a one-way architecture that provides an alternative way for the responses to reach the clients, instead of going through the load balancer, in order to prevent the load balancer from becoming the system bottleneck. There is more information about the implementation of the architecture in OPNET Modeler in the Appendix A.

We have considered 30, 40 and 50 clients in order to compare the results obtained by the six burstiness factors. Each of the tests has been simulated with four seeds for 5, 10, 15 and 20 Web servers in the system.

The workload generated by the set of Web clients contains dynamic and static HTTP

Table 4.1: Workload specification

Number of Web servers	5, 10, 15, 20
Number of clients	30, 40, 50
File size	Lognormal body and heavy tailed
User Think Time (s)	Pareto ($k = 0.3, \alpha = 1.4$)
User session duration (s)	Exponential ($\mu = 600$)
Session Inter-repetition Time (s)	Exponential ($\mu = 30$)

requests to the Web cluster and has been modeled according to recent results in Internet traffic literature. We have considered four types of applications that can be executed concurrently by the Web clients. Each of these applications asks for Web content with a user think time that follows a Pareto distribution ($k = 0.3, \alpha = 1.4$) [17, 30]. As our intention is to stress the system with intense workload by using a low number of clients in order to simplify the scenario design, the equivalent of 30, 40 and 50 clients requesting for Web content with four concurrent application means that an average of 125, 160 and 200 requests per second reach the Web system. The session duration and the session inter-repetition time are modeled according to an exponential distribution, as it is documented in [39] and [23], respectively.

The size of the Web pages the clients ask for has been obtained from the logs of the 24^{th} of June of the 1998 World Cup Web Site [2]. Arlitt *et al.* in [11] estimate, after analysing these logs, that the body of the unique file size distribution follows a lognormal distribution and also that it is heavy-tailed. A summary of the workload specification is shown in Table 4.1.

Two experiments have been carried out in the scenario shown in Figure 4.4. The first experiment stresses the system during a total simulation time of 2000 seconds with the workload specified in Table 4.1. The second experiment considers an increase in the arrival rate at 1000 seconds of the simulation time and for 200 seconds by changing the user think time from a Pareto ($k = 0.3, \alpha = 1.4$) distribution to a Pareto ($k = 0.2, \alpha = 1.4$). This means an important increase in the traffic that arrives at the system. The purpose of this modification is to analyse how the burstiness factors react to a sudden increase of the arrival rate.

4.5.1 First experiment: no changes in the workload

As we want to know the relation between the arrival rate and the burstiness factor detected in each server, we have chosen to compute their correlation by using the standard Pearson method [115]. In the central set of columns, Table 4.2 shows the maximum of the 95^{th} percentile of the correlation values between the arrival rate and the burstiness factors detected in the servers for 30, 40 and 50 clients in this first experiment. We have considered the differences of the values of each statistic in two consecutive slots because this is the way we have formulated most of the expressions above. The correlation values between the arrival rate and the slot frequency are very similar to those presented in this table due to the definition of the slot frequency, which is directly dependent on the burstiness factor, hence we have omitted it.

We find that both statistics are strongly correlated when the used burstiness factor includes the arrival rate in its formula, which is the case of BF2 and BF4. Indeed, BF4 is the most correlated, but its correlation decreases as j increases (in all cases: (i), (ii) and (iii)). The same occurs with BF3. This means that these burstiness factors are probably excessively penalised when j is increased for the proposed workload.

Comparing columns (i), (ii) and (iii) in Table 4.2 for the first experiment we can observe that, despite the fact that its correlation values are less than 0.5, BF1 improves its maximum correlation when more traffic reaches the server. The rest of the burstiness factors do not show this improvement in the first experiment.

In order to obtain a deeper understanding of the benefits of each burstiness factor, Figure 4.5 shows the relation between the differences of the 95^{th} percentile of the arrival rate values and the slot frequency in two consecutive slots for 30 clients. A smooth curve computed by Loess [44] has been added to the plots in Figure 4.5 to highlight the trend of this relation for 5, 10, 15 and 20 servers. As the workload generated by the 30 clients is the same in all the cases, when we have 5 active servers in the Web system, more requests per second arrive at each server than if we have 20 active servers. Hence, the Loess smooth curve differentiates the arrival rate and slot frequency changes for each case.

It can be observed that BF1, BF5 and BF6 (see Figures 4.5a, 4.5i and 4.5j, respectively), scarcely modify the values of slot frequency when changing the arrival rate difference for 5, 10, 15 and 20 servers. The case of BF3 (see Figures 4.5c, 4.5d and 4.5f) shows a bent Loess curve because when a non bursty slot is detected after a sequence of bursty slots,

Table 4.2: Maximum correlation between the 95^{th} percentile of the differences of the burstiness factor and the arrival rate in two consecutive slots for (i) 30 clients (ii) 40 clients and (iii) 50 clients

	First experiment			**Second experiment**		
	(i)	(ii)	(iii)	(i)	(ii)	(iii)
$BF1$	0.3433	0.3840	0.4078	0.2406	0.2417	0.2033
$BF2$	0.8205	0.8195	0.8141	0.7698	0.7263	0.7361
$BF3$						
$j = 3$	0.5386	0.5677	0.5395	0.6940	0.6928	0.6764
$j = 4$	0.5267	0.5187	0.5467	0.7375	0.6848	0.6579
$j = 10$	0.5011	0.4862	0.5273	0.5996	0.6263	0.6550
$BF4$						
$j = 3$	0.8711	0.8508	0.8495	0.8123	0.8287	0.8330
$j = 4$	0.8551	0.8393	0.8469	0.8513	0.8422	0.8390
$j = 10$	0.8225	0.7843	0.7974	0.8160	0.8395	0.8344
$BF5$	0.6313	0.6168	0.5735	0.4497	0.4663	0.4326
$BF6$	0.1179	0.1328	0.0798	0.1585	0.1444	0.1260

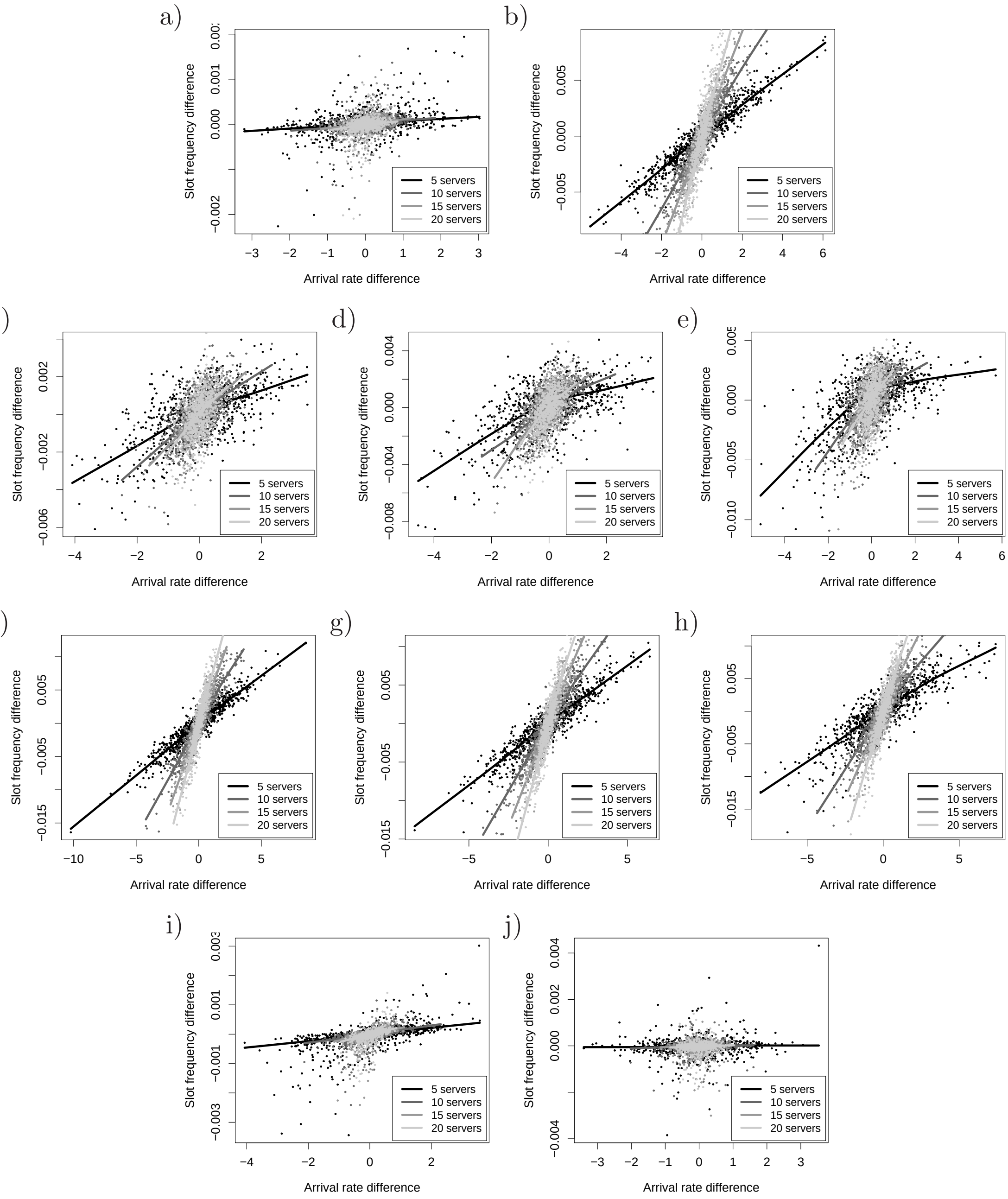


Figure 4.5: First experiment: no changes in the workload for 30 clients. Linear relation between the arrival rate and the frequency of slots considering different burstiness factors: a) BF1; b) BF2; c) BF3 with $j = 3$; d) BF3 with $j = 4$; e) BF3 with $j = 10$; f) BF4 with $j = 3$; g) BF4 with $j = 4$; h) BF4 with $j = 10$; i) BF5; j) BF6.

the burstiness factor decreases suddenly returning to its original value. This can also be observed in Figure 4.2. In the case of BF4, there is also a decrease in the burstiness factor when a non bursty slot is detected, but it is smoother because it depends on the arrival rate detected in the server. The clearer linear relation is obtained with BF2 and BF4 for 5, 10, 15 and 20 servers, which supports the results obtained by Table 4.2.

4.5.2 Second experiment: increasing the workload

In order to contrast the results obtained in the first experiment, we have increased the workload significantly to check whether the correlation values are maintained by the burstiness factors studied in this second experiment. A general reduction can be observed of the correlation values in Table 4.2, in the right set of columns, due to the peak in the arrival rate we introduce when we modify the user think time at 1000 seconds. This is not the case for BF3, which increases its j values and seems to adapt better to this sudden change in the arrival rate. Nevertheless, BF2 and BF4 are still the most correlated.

For a better analysis of the results, we have related the average and peak traffic rates (see Figure 4.6a) as van de Meent *et al.* describe in [134]. The objective is to compare this traffic rate relation with the relation of average and peak burstiness factor values for each of the six burstiness factors considered, highlighting the results obtained for 5, 10, 15 and 20 servers (see Figures 4.6b-k). The more similar the traffic rate relation figure and the burstiness factor relation figure are, the better the burstiness factor represents the variations of the arrival rate and consequently adapts its value.

The comparison shows an excessive penalisation on BF3 and BF4 for all the values of j. The reason is that when a large number of consecutive slots are bursty, then these factors increase their own values more and more. Hence, they easily reach their maximum values (that is 1) and remain near it most of the simulation time. The greater the j of BF3 and BF4 is, the more times they reach 1. This makes it almost impossible for them to detect a sudden, even greater peak in the arrival rate reaching the system because most of the times these factors have reached their maximum values.

BF1, BF5 and BF6 (Figures 4.6b,j,k), respectively) reach their maximum values of 1 fewer times during the simulation time of this second experiment. We can label them as the most conservative because they react to the arrival rate variations by slightly changing their values, which is a drawback if there is a sudden increase in the arrival rate.

BF2 shows almost the same behaviour for 5, 10, 15 and 20 servers (see Figure 4.6c) compared to the variations of the arrival rate plot for a different number of servers (Figure

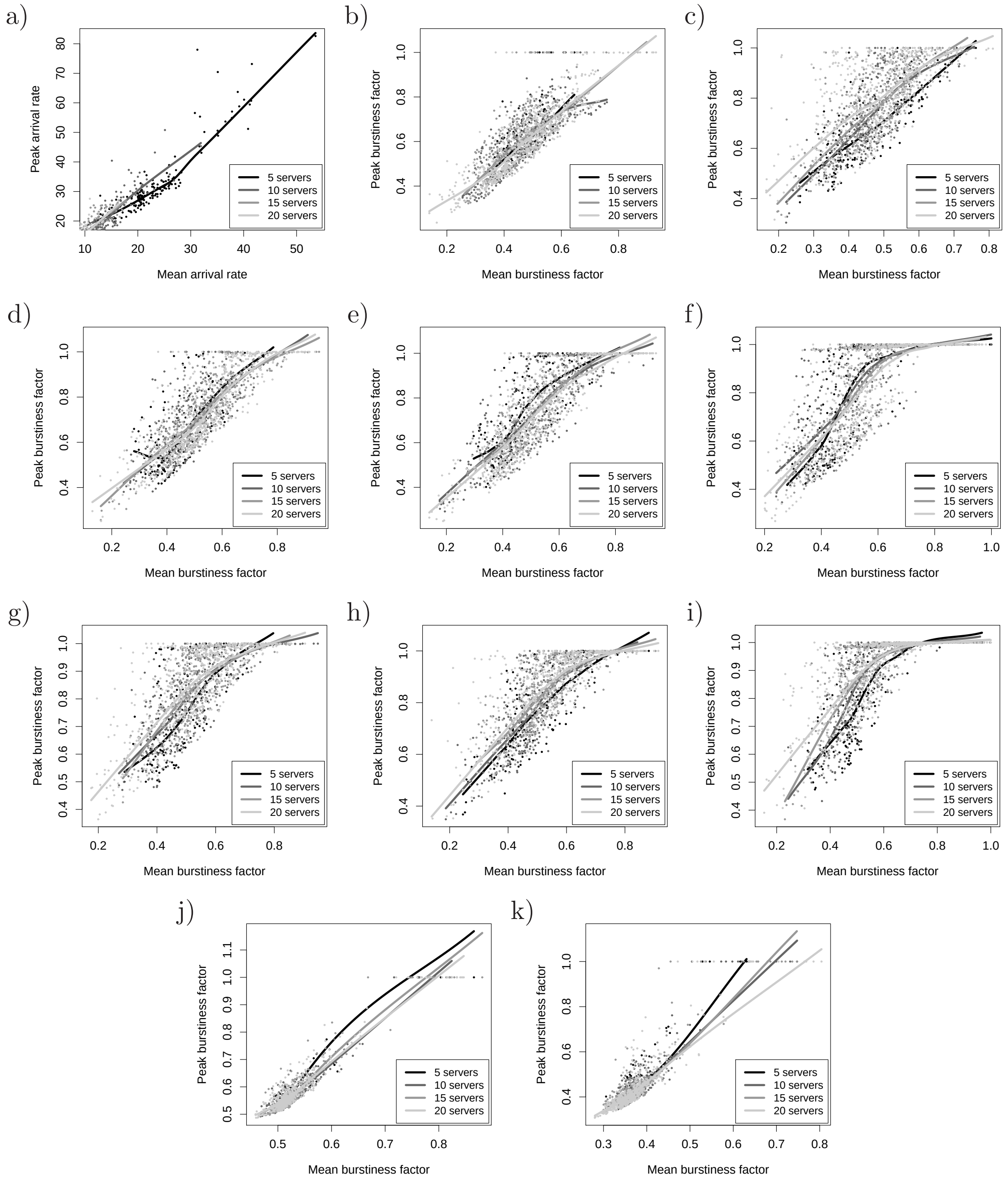


Figure 4.6: Second experiment: increasing the workload for 30 clients. a) Mean and peak traffic rates for each Web server; Mean and peak burstiness factors for each Web server: b) BF1; c) BF2; d) BF3 with $j = 3$; e) BF3 with $j = 4$; f) BF3 with $j = 10$; g) BF4 with $j = 3$; h) BF4 with $j = 4$; i) BF4 with $j = 10$; j) BF5; k) BF6.

4.6a), due to the fact that BF2 adaptively changes with the variations in the workload in each case. However, the comparison among the different number of servers in not very important because their results correspond to different simulations.

4.6 Summary

The aim of this chapter is to study several burstiness factors that detect the variations in the arrival rate at a Web system. This leads to a monitoring scheduling that is defined by adaptive time slot scheduling to minimise the execution overhead of the monitoring itself and the actions of the algorithm that uses the monitored information. If the burstiness factor detects an increase in the arrival rate trend to the system, the adaptive time slot scheduling proposed permits checking times to be closer in the terms of time, or viceversa if an arrival rate decrease is detected. The set of the slot duration is calculated based on the burstiness factor, enabling the adaptive monitoring of the bursty arrivals at the system.

Six different burstiness factors have been considered in this chapter. The main advantage of defining a burstiness factor is that is can be located in any part of the Web system that receives an arrival rate. We have chosen to detect the burstiness in the Web servers of the Web system. Some of the burstiness factors defined (BF1, BF5 and BF6) are conservative in the sense that they do not change their values significantly with the arrival rate variations; while others (BF3 and BF4) include some penalisation when detecting successive bursty slots that force them to change their value considerably. We have found that a burstiness factor that quickly follows changes in the arrival rate trend is good enough as long as its maximum value is not easily reached. Conservative burstiness factors are not very useful if there is a sudden increase in the arrival rate.

An accurate tracking of the arrival rate trend by the burstiness factor is mandatory. It is also important to choose a burstiness factor that permits detecting an increase in the arrival rate independently of the actual workload in the system. These are the main reasons for us to conclude that BF2 seems the best candidate for an adaptive Web system.

Chapter 5

Admission Control and Load Balancing Algorithm

Internet traffic tends to show important growths in demand at certain times of the day or due to special events. The consequence of these traffic peaks is that Web systems that are attending the user demands are congested due to their inability to serve a large amount of requests. The fact that admission control is necessary in these cases is even clearer when QoS is considered in the Web systems. This led us to the necessity of proposing an adaptive algorithm that is capable of adjusting its parameters according to the changes in the arrival patterns.

The main goal of this research is to succeed in the design of a relatively simple algorithm that does not have a high computational cost and that manages the dispatching mechanism with few decisions. These decisions are related to the availability of the Web servers to attend a particular kind of traffic.

We address three main issues in this chapter: firstly, we consider and compare five throughput predictors to be used in a Web system in order to estimate its future performance; secondly, we propose an adaptive QoS-aware admission control algorithm that is based on a resource allocation scheme that includes a throughput predictor and finally, we introduce the load balancing approach considered in the algorithm, which is based on a classical dispatching mechanism.

In order to obtain low overhead, the monitoring of the arrival traffic to the Web system is done following the adaptive time slot scheduling based on the burstiness factor that we defined in the previous chapter. In this chapter, we explain how we also use this scheduling to adaptively invoke the algorithm.

The algorithm is designed to be included in a Web system made up of a set of Web servers locally distributed, which can also form part of a wider geographically distributed load balancing architecture.

In this chapter, we show the benefits of our adaptive time slot scheduling compared to a fixed time slot scheduling [60, 70, 63]. Also a comparison of the behaviour of two classical load balancing policies included in the algorithm is discussed [71, 72]. We detail and discuss the results of the five throughput predictors and the admission control and load balancing algorithm [59, 69]. Finally, we present some results from an implementation of IQRD, the algorithm proposed by Sharifian *et al.* in [123], and comment on the performance differences detected in comparison to our algorithm.

5.1 Introduction

It has been widely studied that Internet traffic is self-similar and that sudden bursts of packets can reach a point in a network infrastructure that offers Web services. This affects the performance of the system considerably when it is not prepared to process that increase in the demand. High variance in incoming traffic and service time distributions can collapse the system in few seconds; therefore it is necessary to control these features by an adaptive algorithm.

We propose an adaptive admission control and load balancing algorithm that prevents the system from a sudden overload by predicting the throughput of the Web servers. We consider five throughput predictors in this work in order to compare their behaviour in our algorithm. The problem of allocating the resources of a Web System that considers QoS and the load balancing strategy is also addressed in this chapter. Our algorithm is prepared to be implemented in a cluster of Web servers to increase the scalability of the solution. Overloaded situations can be solved by this algorithm, which guarantees the satisfactory performance of the system by controlling the utilisation level of the Web servers of the cluster.

An important contribution of our work is the adaptive overhead our solution introduces in the system. It is critical to avoid checking the system continuously, i.e. at each incoming request because this produces an enormous overhead in the front end of the system, but it is also risky to check the system during fixed intervals because a sudden increase may not be detected until the system is already overloaded. We have analysed in Chapter 3 the most important related work in order to know how other authors handle or control overhead, and we have observed that very few works propose methods to reduce the overhead.

The following sections of this chapter are organised as follows:

- The steps we take in order to obtain a low overhead are detailed in Section 5.2.
- Section 5.3 introduces an overview of the algorithm.
- The throughput predictors we have considered are described in Section 5.4.
- Section 5.5 details the resource allocation strategy.
- The load balancing strategy is covered in Section 5.6, including the description of the two classical load balancing policies that we have considered to include in our algorithm.
- Results obtained are detailed in Section 5.7. They show the benefits of the adaptive time slot scheduling and a comparison of the performance obtained by the five throughput predictors in the admission control and load balancing algorithm. The behaviour of two classical load balancing policies included in the algorithm is studied as well. Also a comparison of our algorithm with the algorithm proposed in [123], IQRD, is described in this section.
- Finally, we summarise the chapter.

5.2 Aiming for low overhead

We propose an admission control and load balancing algorithm that is based on throughput prediction for a Web system. In order to achieve a low overhead of the algorithm, we plan the invocation times based on the arrival rate. We defined a burstiness factor, $b(k)$, in the previous chapter. This factor varies its value in a range of [0,1] and gives an indication of the burstiness perceived in the entry point of the system. Based on this factor, we defined an adaptive time slot scheduling in the previous chapter that sets the frequency the Web system asks for the monitored metrics. We are going to use this adaptive time slot scheduling to set the times the admission control and load balancing algorithm is invoked, as well.

Indeed, when the algorithm is invoked, it needs to ask for the monitored information in the server nodes of the Web system.

Therefore, we divide the time into slots (k) of different durations ($d(k)$) during the experiment. The burstiness detected in the system influences the execution of the algorithm in this way: when an increase in the burstiness is detected, then the algorithm is invoked more frequently, and viceversa.

There are two options to invoke an algorithm, as we previously described in Chapter 3. Supposing the cost of executing the algorithm is $O(x)$ and the total observation time is T, let us then analyse the cost each option has:

- **Periodical invocation frequency**: In this case, the algorithm is executed periodically and the decisions taken are maintained until the next execution of the algorithm. The total observation time T can then be divide into n periods or slots. The approximate cost of this case would be $O(n \cdot x)$.

- **Non-periodical invocation frequency**: Normally this means that the algorithm is executed each time a new request or session arrives to the system. In this case, and considering that r requests are received during the whole observation time, the cost of the algorithm would be $O(r \cdot x)$, r being several orders of magnitude greater than n.

  Adaptive scheduling can also be included as non-periodical scheduling. This is the case of [133, 19] that vary the invocation method depending on the load, but only under some circumstances, as we have previously commented on in Section 3.5.

We propose a variation of the frequency of the execution of the algorithm in all cases, always depending on the burstiness detected in the system.

5.3 Algorithm overview

The admission control and load balancing algorithm is based on throughput prediction for a Web system. We have defined five throughput predictors that give us the trend of the system behaviour and permit the algorithm to take decisions that maintain the performance of the system. Different classes of requests with different priorities are considered in this work. The Service Level Agreement (SLA) of the requests is defined in terms of CPU utilisation of the Web servers. We show in this section an overview of the characteristics of the algorithm. First, we introduce the system architecture that is proposed for the algorithm. We also describe the QoS introduced in the system by defining the SLA of the requests. And finally, we discuss the performance metrics considered by the algorithm.

5.3.1 System Architecture

The system architecture proposed is based on Web cluster-based network servers and includes a front-end Web switch. A layer-7 Web switch is normally described as a content-aware switch that can de-encapsulate the requests up to the application level and classify them on the basis of this information, but as we described in Chapter 2, it can easily be the bottleneck of the Web system. Other authors [14, 109, 83, 38, 144] have already solved this problem by transferring the request distribution mechanism to the back-end nodes and replacing the content-aware Web switch with a content-blind Web switch.

Another problem of a content-aware Web switch is the distribution of requests based on HTTP/1.1 persistent connections, that has also been solved in previous proposals by other authors [12, 93, 125, 95, 97]. Hence, we consider our one-way content-aware architecture includes one or more distributor nodes, leaving the underlying TCP implementation open to implement one of these proposals.

The cluster of Web servers is locally connected to the Web switch in a two-tier organisation (Web server and App/DB server). Each Web server attends the requests that ask for static files, namely static requests and the App/DB server is accessed when the request asks for a Web page that needs to retrieve dynamic content (dynamic requests).

The Web cluster can be considered either a part of a wider system that geographically distributes the load or an Autonomous System (AS).

5.3.2 QoS-awareness

Different classes of requests are distinguished in the algorithm. We have already described that a distinction between static and dynamic requests is considered as they mean different loads in the two tiers of the server's architecture. A QoS differentiation in the requests is also introduced. We define the priority of each class of requests by setting a fraction of the utilisation of the whole Web system for each class. Therefore, we consider a set of classes, $C = \{c_1, c_2, \ldots, c_r\}$, and define for them a normalised utilisation value in a decreasing order. Hence the class of requests that represent c_1 have more priority than the class c_2, and so on. The sum of the utilisation values of all the classes is equal to 1, which represents the whole utilisation of the Web system. The admission control and load balancing algorithm dynamically defines how the different classes of requests are distributed among the Web servers.

5.3.3 Performance metrics used in the algorithm

It is important to detect the resource that can be the bottleneck of the Web system and hence, determine the performance metrics that will manage the admission control and load balancing algorithm. As we have previously described in Section 3.4, many authors have considered the CPU utilisation as a metric to be checked in the admission control mechanism [108, 6, 15, 39, 136, 52, 8, 152, 84, 123]. Indeed, it has been detected that the main bottleneck of a Web system is the CPU when dynamic content is requested [119]. Hence, we have considered the CPU utilisation as the main metric to estimate in order to control the performance of the Web system. Other performance metrics have to be monitored to estimate the CPU utilisation; some of them can be obtained from the front-end of the Web system, but others have to be requested to the Web and App/DB servers. The monitoring results obtained have to be transferred to the distributor node/s.

Let us describe the metrics that we need to monitor to execute our algorithm:

- The arrival rate, that is used to obtain the burstiness factor and hence, to set the adaptive time slot scheduling. The arrival rate can be easily monitored by the front-end of the Web system, counting the requests that arrive during a slot. The arrival rate for a slot k, $\lambda(k)$, is obtained by dividing the total number of incoming request by the slot duration, $d(k)$.

- The service times needed to process the static and dynamic requests are obtained from the Web and App/DB servers. We obtain the service time average at each slot, $\delta(k)$, and use it in order to estimate the utilisation of the Web and App/DB servers.

- The average throughput for a slot k, $x(k)$, is also obtained from the Web and App/DB servers, and used to estimate the throughput of the Web and App/DB servers during next slot. The throughput is also used to control the error in the throughput prediction.

- The average CPU utilisation of the Web and App/DB servers, $u(k)$, that is used as a factor in the expression of some throughput predictors, and also used in order to control the error of our prediction.

We obtain the service times and adjust the prediction of the utilisation in the servers based on the predicted throughput because it is known that the service times increase when the server starts to be overloaded [35, 52].

The different metrics are monitored in the Web and App/DB servers, but only transmitted to the distributor node/s and used when the admission control and load balancing algorithm is invoked.

5.4 Throughput Prediction

First of all, our algorithm is based on throughput prediction. Thus, in this section five throughput predictors are described. A previous version of the predictors P1-P3 were introduced in [59]. We have extended the research in throughput prediction in order to achieve better results. Let us present these five predictors.

5.4.1 P1: based on filtering

This predictor is a moving average between the estimated value of the throughput in the last slot and the harmonic mean of the real throughput measured in the last two slots.

$$\hat{x}_1(k+1) = (1 - a(k+1)) \cdot \hat{x}_1(k) + a(k+1) \cdot \frac{2}{\frac{1}{x(k)} + \frac{1}{x(k-1)}} \tag{5.1}$$

$a(k+1)$ being the probability that balances the weight of the expression in function of the duration of the next slot, $d(k+1)$; hence, it indirectly depends on the burstiness factor:

$$a(k+1) = \tfrac{2 \cdot T - d(k+1)}{2 \cdot T + d(k+1)}$$

Therefore, this estimated throughput depends on two terms; the last computed throughput prediction and the average of the actual and previous throughput measurements. The result of this computation is filtering throughput values based on the probability $a(k+1)$. Since the duration of the slot $k+1$ depends on burstiness, the weight of $a(k+1)$ indirectly depends on burstiness too. The factor $a(k+1)$ normally places more weight upon previous throughput measured in the servers during the slots k and $k-1$ than on the previous throughput estimation, unless the value of $d(k{+}1) > \frac{2}{3}{\cdot}T$. The smaller is the duration of the slot, the more weight the previous throughput measures have in this predictor. The main effect of this estimation is smoothing traffic peaks in order to hold an accurate performance estimation of the servers in the long run.

5.4.2 P2: based on burstiness

This predictor includes the burstiness factor we have described in Chapter 4. It is included in order to factorise the tendency of the burstiness during the last two periods with the throughput variation as a factor $\beta(k+1)$:

$$\beta(k+1) = (b(k) - b(k-1)) \cdot |x(k) - x(k-1)| \tag{5.2}$$

This factor considers the throughput variation during the last two slots. Thus, it measures the difference between the current and the previous burstiness factors, and then multiplies it by the absolute value of the difference between measured throughputs at slots k and $k-1$. The resulting product expresses the amount of variation of the servers' performance due to the burstiness on client transaction arrivals.

This factor is then multiplied by the server utilisation in order to scale the increase or decrease of previous slot throughput:

$$\hat{x}_2(k+1) = x(k) - (\beta(k+1) \cdot u(k)) \tag{5.3}$$

The goal of this estimator is to decrease the throughput prediction when a burst of requests is detected, depending on the current utilisation of the servers.

5.4.3 P3: based on filtering and burstiness

The harmonic mean of the predictor P1 and P2 is considered as the third predictor in order to balance the effect predictors P1 and P2 may have in the system.

$$\hat{x}_3(k+1) = \frac{2}{\frac{1}{\hat{x}_1(k+1)} + \frac{1}{\hat{x}_2(k+1)}} \tag{5.4}$$

5.4.4 P4: based on Least Mean Square (LMS)

We have also considered the LMS algorithm [140, 76] to predict the throughput. LMS introduces an iterative procedure that makes successive corrections to a weight vector that minimises the mean square error. This filter has been previously used in throughput prediction by Garroppo *et al.* in [57].

Let $\underline{w}(k+1)$ denotes the weight vector of the LMS filter that is computed at the k slot. The operation can be expressed by the following recursive operation:

$$\underline{w}(k+1) = \underline{w}(k) + \mu \cdot [x(k) - \hat{x}_4(k)] \cdot \underline{x}(k) \tag{5.5}$$

where M is the number of tap weights used in the adaptive transversal filter, μ is the step-size parameter and the vectors $\underline{w}(k)$ and $\underline{x}(k)$ are defined as:

$$\underline{w}(k) = [w_0(k), w_1(k), ..., w_{M-1}(k)]^T$$

$$\underline{x}(k) = [x(k), x(k-1), ..., x(k-M+1)]^T$$

The predicted value of throughput is obtained by linear prediction with this expression:

$$\hat{x}_4(k+1) = \sum_{x=0}^{M-1} w(x) \cdot x(k-x) \tag{5.6}$$

5.4.5 P5: based on Normalised Least Mean Square (NLMS)

A normalised version of the LMS filter was proposed in [74] to avoid the sensitivity of the LMS algorithm to the scaling of its input $x(k)$. The solution was to normalise the previous expression by dividing the vector $\underline{x}(k)$ by the square of its Euclidean norm.

$$\underline{w}(k+1) = \underline{w}(k) + \mu \cdot [x(k) - \hat{x}_5(k)] \cdot \frac{\underline{x}(k)}{\|\underline{x}(k)\|^2} \tag{5.7}$$

5.4.6 Throughput prediction results

We have prepared a simulation experiment in OPNET Modeler [3] in order to depict the behaviour of these predictors. The results are shown in Figure 5.1. We have included

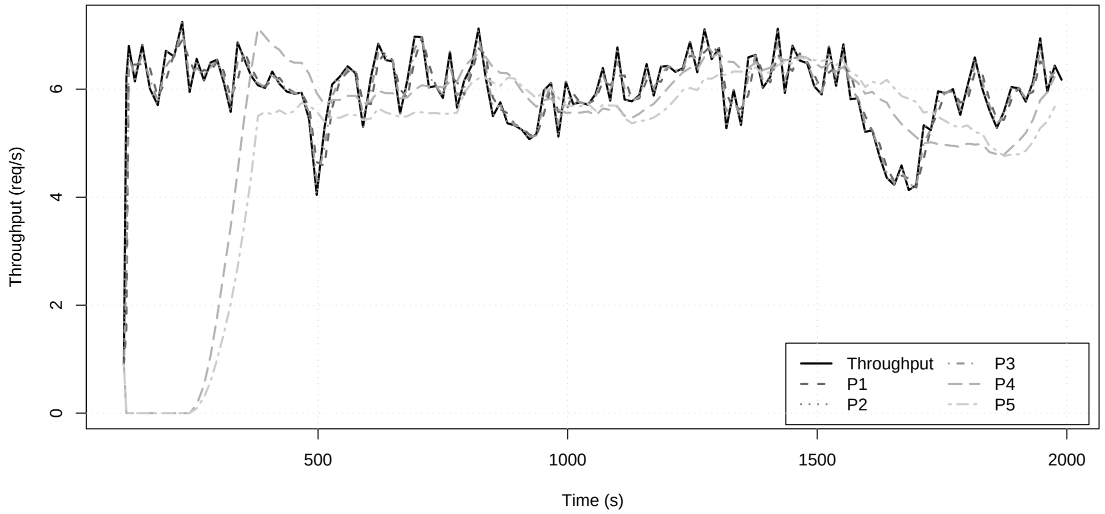


Figure 5.1: Throughput predictions

the throughput of a server and the five throughput predictions introduced above. It can be observed how the predictors P1-P3 follow the throughput from the very beginning of the simulation. As the predictors P4-P5 are based on filter theory, they need more slots in order to obtain a good estimation. In the case of the simulation results presented in Figure 5.1, they need almost 400 seconds. This is due to the number of tap weights (M) has been set to 20, hence during the first 20 slots there is no throughput prediction. After the 21st slot, the prediction slowly moves during another 20 slots to an acceptable value.

Nevertheless, it is very difficult to check the effectiveness of the predictor visually in Figure 5.1. In order to know how reliable the predictions are, we have computed the mean squared error they introduce in the admission control and load balancing algorithm. The complete set of results is described in Section 5.7.

Thus, let us describe first the resource allocation part of the algorithm and the load balancing approach in order to depict the whole picture.

5.5 Resource Allocation

Throughput prediction enables us to estimate the maximum utilisation allowed in the servers for each service class, and then to set a maximum number of accepted requests in order to limit the utilisation of the servers and avoid a possible congestion situation, while guaranteeing the QoS. An earlier version of the resource allocation described in this section was presented in [62].

With the value of one of the throughput predictors described in previous section, $\hat{x}(k+1)$, and the service time, $\delta(k)$, obtained from the servers we can make an approximation of the utilisation level the servers will have in the following slot for each class of service. An adjustment of these utilisation levels is necessary in order to fulfil the SLA. Once we obtain the adjusted utilisation level the servers should not exceed, we can calculate the maximum number of requests each server can process of each class of service.

Hence, as a first step, we predict the utilisation that each class of traffic will have in each server of the Web system with this expression:

$$\hat{u}(k+1) = \hat{x}(k+1) \cdot \delta(k) \tag{5.8}$$

This prediction does not include the SLA we have defined for each class of requests. We then normalise the predicted utilisation to guarantee the priority requirements of each class. We need to add some subscripts to the expressions because the throughput and utilisation predictions are calculated for each Web and App/DB server, and for each class of service. Therefore, we include three subscripts:

- The first subscript indicates the number of the Web server and App/DB server set, that is represented by $i \in \{1, \ldots, N\}$.
- The second subscript states the class of service and it is represented by $j \in \{1, \ldots, r\}$.
- The third subscript points out whether the content requested is static or dynamic, $z \in \{sta, dyn\}$. In case the content is static, then it will be attended by the Web server. The App/DB server will carry out most of the processing work if the content requested is dynamic. Hence, we compute the utilisation separately in the two tiers of the Web system architecture.

The normalisation of the utilisation estimation is done to distribute the 100% of the available capacity of the Web system between all the classes of requests, N being the number of Web and App/DB server sets that are included in the Web system:

$$\hat{u}^{'}_{i,j,z}(k+1) = (c_j \cdot N)\,\frac{\hat{u}_{i,j,z}(k)}{\sum_{\forall i}\hat{u}_{i,j,z}(k)} \tag{5.9}$$

With this normalisation, we assure that each traffic class has reserved the utilisation of the Web system that corresponds to its SLA. In order to know the maximum number of requests that are going to be accepted for each traffic class in each server during the following slot, we make the inverse operation and multiply the obtained throughput by the duration of the next slot:

$$\gamma_{i,j,z}(k+1) = \frac{\hat{u}^{'}_{i,j,z}(k+1)}{\delta_{i,z}(k)} \cdot d(k+1) \tag{5.10}$$

During the next slot, the distributor counts the number of accepted requests of each class in each Web and App/DB server. When $\gamma_{i,j,z}(k+1)$ is reached, then the server stops attending that class of requests until the next invocation of the admission control and load balancing algorithm.

We can then consider that each server has a margin of unused utilisation left at the end of each slot that is in the range of values $\left[0, \hat{u}^{'}_{i,j,z}(k+1)\right]$. The computation of the margin utilisation is done by the subtraction of the reserved utilisation and real utilisation of the slot $k+1$: $\hat{u}^{'}_{i,j,z}(k+1) - u_{i,j,z}(k+1)$.

5.6 Load Balancing mechanism

We have designed a dispatching mechanism that is based on classic load balancing policies, that are used to distribute the incoming requests among the Web and App/DB servers, distinguishing among the different classes of service requested. The distributor node performs a content aware analysis to learn which service class is needed to attend each request. r being the number of service classes, we define $2 \cdot r$ independent classical load balancing policy instances; r corresponds to the static requests, and r for the dynamic requests (that normally imply a higher computational cost in the App/DB server).

One of the classic load balancing policies selected is Round Robin (RR). In this case, the distributor node organises the Web and App/DB servers in a cyclical manner for each class of service. So, if there are N servers and all the servers are *active* for attending requests that ask for a particular class of service, each server $i \in 1,..,N$ will receive the request number x, if $i = (x \bmod N) + 1$. Note that it is possible that not all the servers are *active*. If a Web server or App/DB server reaches the maximum number of requests for a class of service (j) that was set in the previous invocation of the admission control algorithm, then it is *disabled* from the possible selection by the distributor node until the next algorithm's invocation. Therefore the rejection of requests begins when all the Web or App/DB servers of the system are disabled for a class of service.

We also consider the Least Connection (LC) policy in the distribution of the requests among the servers. In this case, the server that is selected to attend a request is the one with the least number of connections. As before, when servers reach the maximum number of requests of service class j, then the load balancer automatically changes the server status to *disabled*. Therefore the distributor stops sending to those servers requests that ask for service class j until the next algorithm's invocation.

5.7 Simulation Results

This section includes all the simulation results we have obtained with the implementation of the algorithm. We have implemented our algorithm in the simulation tool OPNET Modeler [3] which permits it to simulate accurately the layers of the TCP/IP stack. Hence, we have considered a realistic simulation model including all the protocols below in the TCP/IP stack that take place during an application-application level communication. The routing protocol considered at network level is RIP in all the routers of the network. We have made modifications to some of the OPNET Modeler standard models to adapt them to our approach. The details of this implementation are shown in the Appendix A. We divide this section into four different subsections:

- To start with, we configure two classical load balancing policies, RR and LC, in our distributing mechanism and compare their performance with different Web cluster sizes. The results of this comparison are shown in Subsection 5.7.1.

- In second place, we implement the five throughput predictors defined previously in OPNET Modeler and, after executing the algorithm with two different workloads, we show and discuss the results obtained in Subsection 5.7.2.

- We analyse the benefits of the adaptive time slot scheduling comparing it with the execution of the algorithm using periodical time slot scheduling in Subsection 5.7.3.

Table 5.1: Workload specification (for each service) for the classical load balancing comparison

Number of Web servers	5, 10, 15, 20
Number of clients	15, 30
Objects size (bytes)	Lognormal ($\mu = 5000, \sigma^2 = 1000$)
Number of objects per page	Pareto ($k = 20, \alpha = 1.4$)
User Think Time (s)	Pareto ($k = 1, \alpha = 1.4$)
User session duration (s)	Exponential ($\mu = 600$)
Session Inter-repetition Time (s)	Exponential ($\mu = 30$)

These results show that our adaptive time slot scheduling outperforms a static approach and permits the system to better react to sudden increases in the arrival rate.

- And finally we compare our algorithm to the admission control and load balancing mechanism proposed by Sharifian *et al.* in [123], and discuss the results in Subsection 5.7.4.

The workload we have used in each of the simulations in order to obtain the results shown in the different subsections is slightly different, this is why we detail the workload used in each of the subsections trying not to be very repetitive. We specify the workload in the form of a table, so it is easy to compare the workloads of the different results reported in this section.

There are some characteristics that all the simulations have in common. We consider two different service classes, named c_1 and c_2, in all the simulations. Each service class contains two types of applications: one that asks for dynamic content and another that asks for static content. Static requests are attended by the Web servers while dynamic requests require access to the App/DB server. Hence, we define in the system four types of applications or services (c_1 static, c_1 dynamic, c_2 static and c_2 dynamic) that execute concurrently in the Web system and whose service is asked, also concurrently, by the Web clients. The SLA specified for each service class is: $c_1 = 0.625$ for class-1 requests and $c_2 = 0.375$ for class-2. Requests asking for both classes of service are introduced in the system in the same proportion. We refer indistinctly to traffic class, service class or request type in order to indicate a class of service. We have repeated every single simulation four times with four different seeds during a total simulation time of $T = 2000$ seconds.

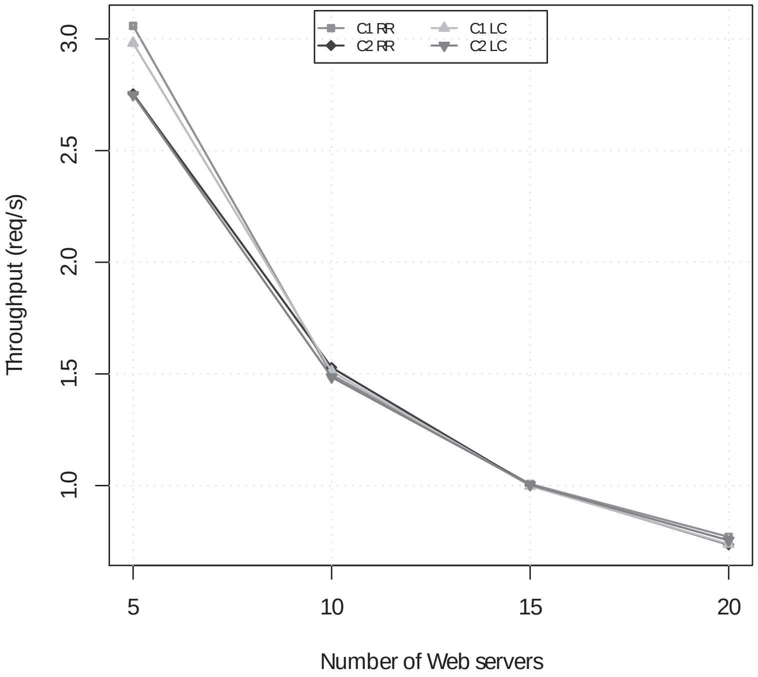


Figure 5.2: 95^{th} percentile of the throughput of dynamic requests for 15 clients

5.7.1 Classical Load Balancing Policy Selection

The workload used in this comparison is specified in Table 5.1. We have simulated in OPNET Modeler a Web cluster made up of 5, 10, 15 and 20 Web and App/DB servers in order to compare the performance of the load balancing policies with different sizes of the Web cluster.

The workload is generated by a set composed of 15 or 30 clients that sends requests to the Web cluster and receives responses with an inter-arrival time that follows a Pareto distribution. The Web pages the clients ask for include a number of objects that also follows a Pareto distribution and the object size is modelled according to a lognormal distribution [30].

Let us analyse first the results obtained with 15 clients. Both classical policies, RR and LC, have been simulated in a cluster with 5, 10, 15 and 20 servers. The static requests are all attended in the four different cluster sizes. Hence, dynamic requests load the App/DB servers more than the static ones load the Web servers (as we have previously commented on in Subsection 5.3.3). Figure 5.2 represents the 95^{th} percentile of the throughput of dynamic requests of class-1 and class-2 obtained by each of the servers, which is very similar when

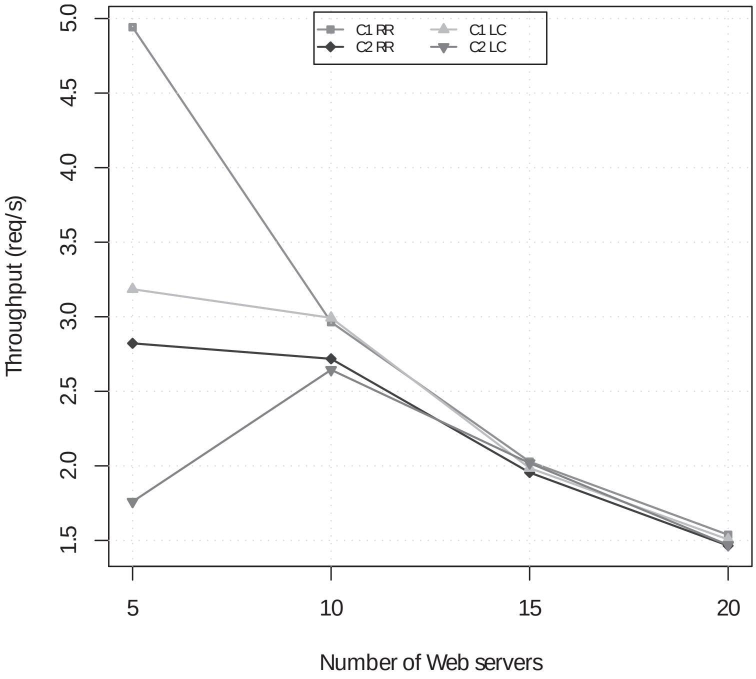


Figure 5.3: 95th percentile of the throughput of dynamic requests for 30 clients

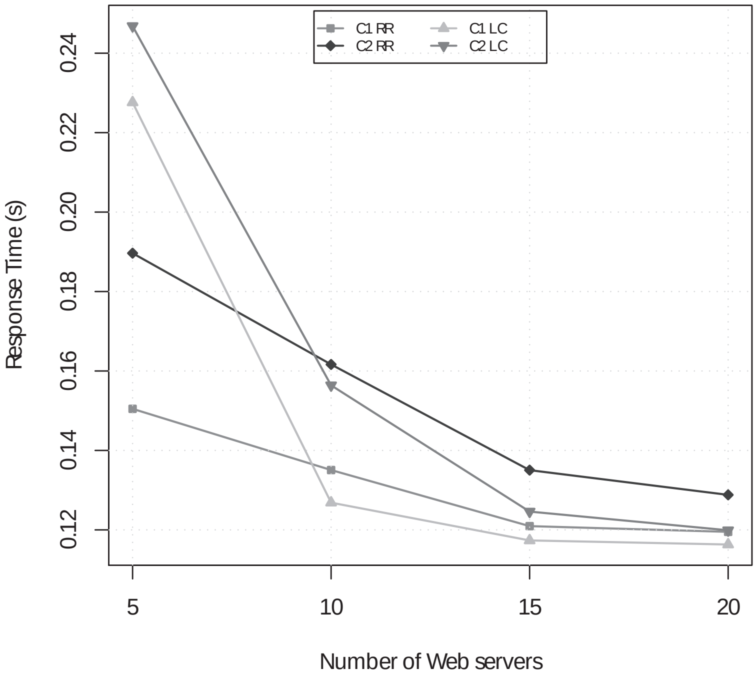


Figure 5.4: 95th percentile of the response time of dynamic requests for 30 clients

using the LC and RR policies. We can observe a little difference in the throughput between the two classes of traffic in the case of a cluster composed of 5 servers. The larger number of rejections of class-2 requests with 5 clients when using the LC policy cause this throughput difference.

Regarding the 30 clients example, the results are quite different because now there are more requests arriving to the Web system and the servers are more loaded. Figure 5.3 shows the 95^{th} percentile of the throughput of dynamic requests of class-1 and class-2 obtained by each of the servers. In fact, it can be observed that there is a better behaviour for the RR policy in the case of 5 servers than the LC policy. The reason is implicit on the load balancing strategy used. When there are few servers left in an *active* status that can attend requests asking for a service class, the load balancing mechanism chosen becomes crucial to ensure the performance of the system. If the RR policy is used, then all the remaining servers will receive the same number of requests. But when the LC policy is used, the *active* server with the least number of connections will receive all the requests until its number of connections exceeds the number of connections of other *active* server. This can be a drawback because the number of connections among servers since the beginning of the simulation may be quite different and, in the case that only two servers remain *active*, it might happen that in less than a second the server with the least number of connections ends up overloaded because all the requests are sent to it. When the cluster includes more than 5 servers both policies get similar results. The 95^{th} percentile of the response time of dynamic requests obtained by both load balancing policies with 30 clients is represented in Figure 5.4. We can observe that the response time obtained by the RR policy is lower than the response time obtained by the LC policy for 5 servers.

Hence, in this subsection we have compared the RR policy and the LC policy in our algorithm, detecting that the first outperforms the latter one when 5 servers or less are *active* in the cluster .

5.7.2 Throughput Prediction Comparison

The five throughput predictors described in Section 5.5 have been tested in our admission control and load balancing algorithm. We configure a simulation scenario in OPNET Modeler that consists of 5 Web and App/DB servers that attend the two different classes of requests, c_1 and c_2. The workload is generated in the Web system by 30, 40, 50, 60, 70, 80,

Table 5.2: Workload 1 specification (for each service) for the throughput prediction comparison

Number of Web servers	5
Number of clients	30, 40, 50, 60, 70, 80, 90, 100
File size	Lognormal body and heavy-tailed
User Think Time (s)	Pareto ($k = 0.3, \alpha = 1.4$)
User session duration (s)	Exponential ($\mu = 600$)
Session Inter-repetition Time (s)	Exponential ($\mu = 30$)

90 and 100 Web clients, as we are interested in stressing the system to test the algorithm with an increasingly high workload. So, the Web system starts rejecting requests when it is overloaded.

We configure two different workloads in order to test the algorithm more accurately. Both are basically the same, the only difference is in the user think time. Let us describe these two workloads independently.

Workload 1 The first workload specification used in this comparison is described in Table 5.2. We consider a Pareto user think time in the Web clients, and the session duration and the session inter-repetition time are modelled according to a exponential distribution. The file size has been obtained from the logs of the 24th of June of the 1998 World Cup Web Site [2]. Arlitt et al. in [11] estimate, after analysing these logs, that the body of the unique file size distribution follows a lognormal distribution and also that it is heavy-tailed.

Workload 2 The second workload is basically the same as the first one, but with one difference. In order to configure a more variable workload, we insert traffic peaks every 300 seconds. Hence, we modify the user think time for 30 seconds in order to provide more burstiness in the simulation. After each 300 seconds of the workload specified in Table 5.2, we change for 30 seconds the user think time to Pareto ($k = 0.2, \alpha = 1.4$). This means a rise in the arrival rate after certain time of “stability” that permits us to test the algorithm under more severe circumstances.

The representation of the arrival rate of the Web requests generated with both *Workload 1* and *Workload 2* is depicted in Figure 5.5.

As we introduce the same proportion of static and dynamic requests in the system and

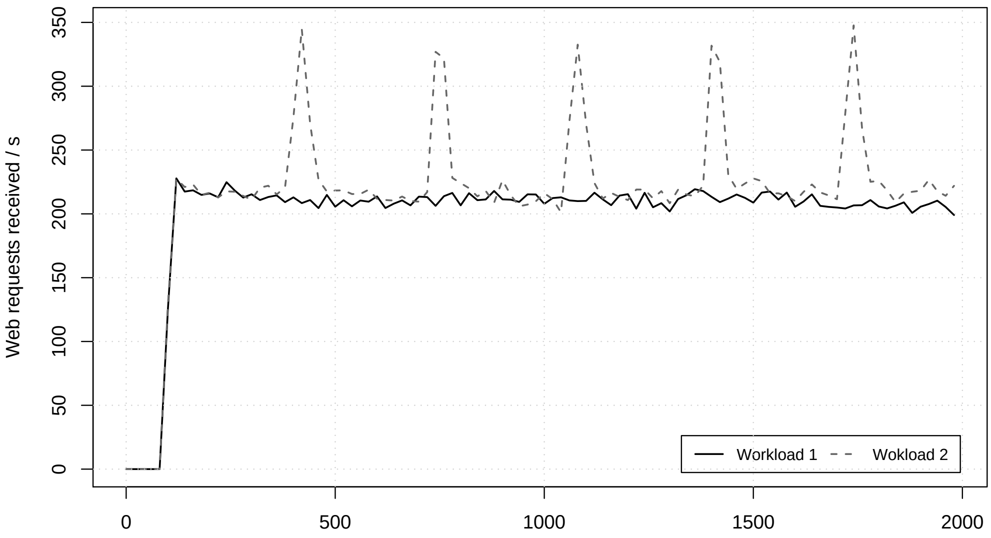


Figure 5.5: Workload 1 and Workload 2

the admission control algorithm reserves some CPU utilisation for each service type, the first components of the Web system that become overloaded are the App/DB servers. Therefore, the bottleneck of the architecture are the App/DB servers as we will observe in the results reported in this subsection.

In order to check this, we have to analyse the utilisation level results, but first, let us have a look at the error in throughput predictions [61]. We have computed the mean squared error of the throughput predicted in a slot and the throughput monitored in the next slot in order to compare the effectiveness of the predictions made by the five predictors we defined in Section 5.4:

$$E_{ms} = \sqrt{\frac{\sum_{k=1}^{n} \left(x_{i,j,z}\left(k\right) - \hat{x}_{i,j,z}\left(k\right)\right)^2}{n}} \tag{5.11}$$

The computed error for each prediction is shown in Figures 5.6 and 5.7 for static requests, and in Figures 5.8 and 5.9 for dynamic traffic, and both workloads. We have used the same scale for both static and both dynamic traffic results in order to compare them.

Considering static requests, it can be observed that the error in the prediction grows with the number of clients in both cases (it increments more in the case of *Workload 2*). This is due to the fact that the throughput values increase with the number of clients because static traffic is not rejected in any simulation.

The case of the prediction errors in dynamic requests' throughput is different than static, as it is limited by the admission control algorithm. Hence, the prediction error is reduced at some point. On one hand, we can observe in the upper part of Figure 5.8 that the throughput errors of all predictors for class-1 dynamic requests increase slightly until 70 clients and then, they are abruptly reduced to a value near to zero. On the other hand, we notice in the bottom part of the figure, the decrease of the class-2 throughput starts at 50 clients.

The throughput prediction error results for *Workload 2* are represented in Figure 5.9. In this case, we can observe that it decreases with the number of clients for both class-1 and class-2 traffic. This decreasing tendency is directly related to an increasing number of rejections carried out by the algorithm as the number of clients grows larger.

While requests are not rejected, the throughput prediction error tends to increase as the throughput increases. However, when rejections start, the value of the prediction error remains near zero because the throughput is controlled by the algorithm, hence, the predictions are more precise.

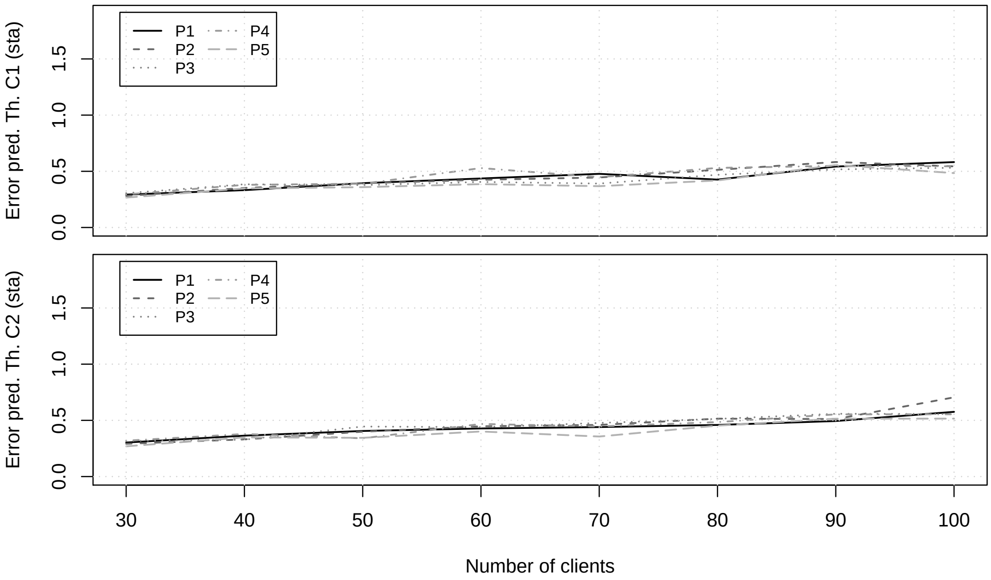


Figure 5.6: Workload 1: Mean squared error of static requests' throughput predictions

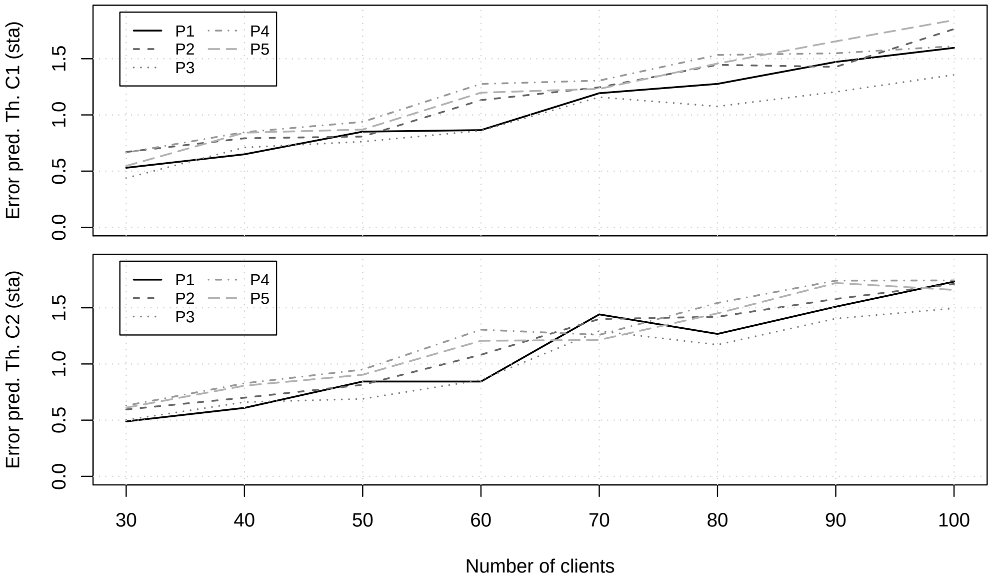


Figure 5.7: Workload 2: Mean squared error of static requests' throughput predictions

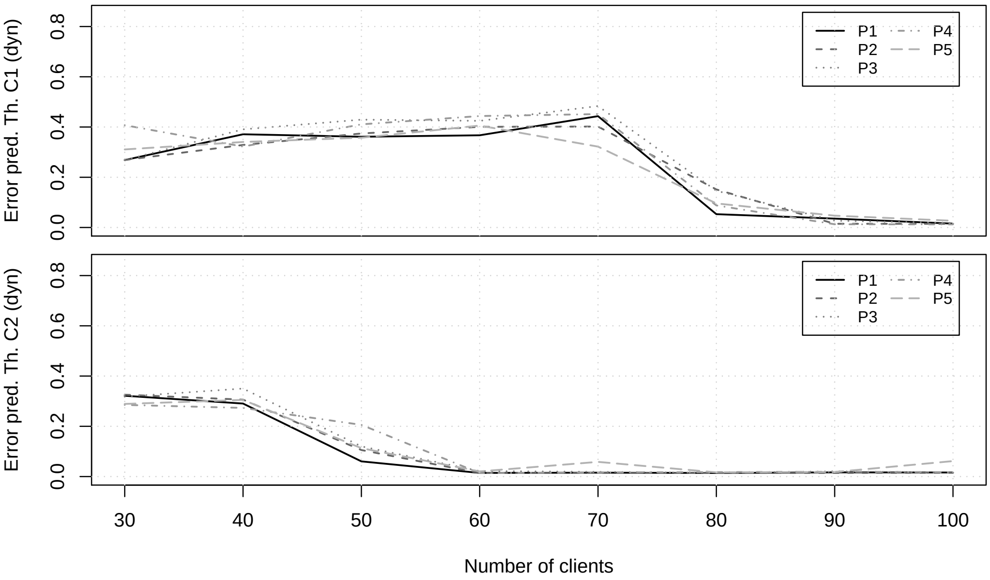


Figure 5.8: Workload 1: Mean squared error of dynamic requests' throughput predictions

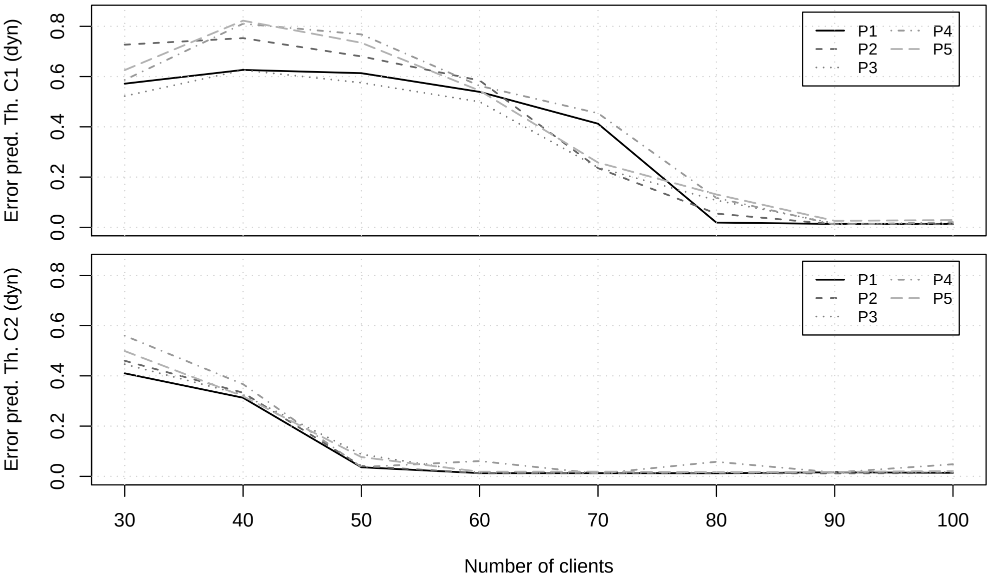


Figure 5.9: Workload 2: Mean squared error of dynamic requests' throughput predictions

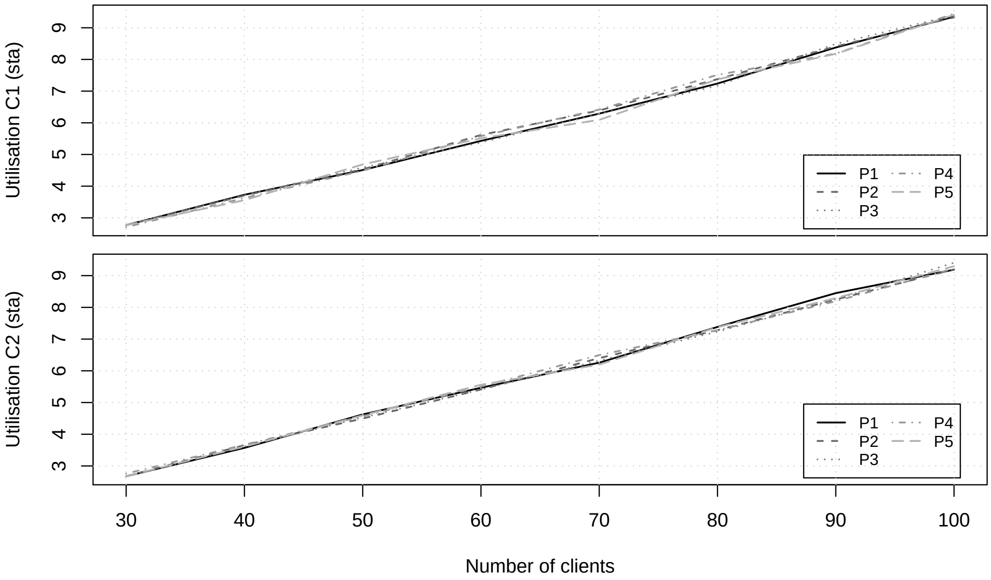


Figure 5.10: Workload 1: 95^{th} percentile of the average Web server utilisation for static requests

Let us continue studying the 95^{th} percentile of the average CPU utilisation of the Web servers and the App/DB servers separately. Figure 5.10 represents the Web server's CPU utilisation. Considering that the maximum utilisation is 100%, we observe that the Web server CPUs are not overloaded with *Workload 1.* We have omitted the results of *Workload 2* because comparatively they increase slightly (up to 12%), but basically are very similar.

The 95^{th} percentile of the average CPU utilisation of the App/DB servers is shown in Figure 5.11 for *Workload 1* and Figure 5.12 for *Workload 2.* Here we can see a difference between both figures as *Workload 2* means more traffic in the system. Furthermore, we can also observe the differentiation of service classes. *Workload 1* makes the App/DB servers be at 100% of their reserved utilisation level for class-1 with 70 clients and *Workload 2* with 50 clients. The maximum utilisation for class-2 requests is almost reached with 40 and 30 clients, respectively.

We can also observe that the average CPU utilisation achieved by the App/DB servers guarantees the SLA defined in the algorithm. For class-1 traffic, $c_1 = 0.625$ (62.5% of utilisation of the servers) and for class-2 $c_2 = 0.375$ (37.5%).

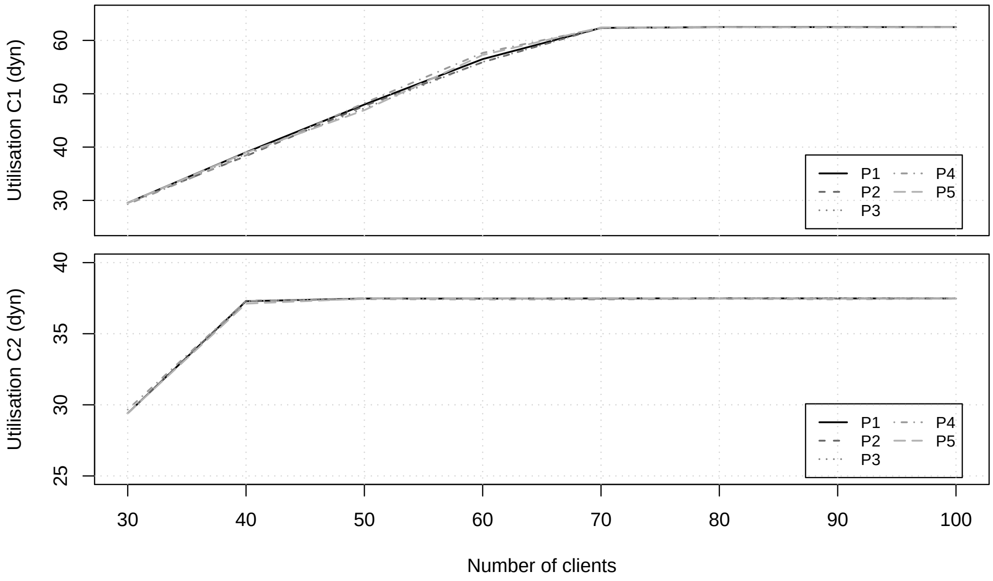


Figure 5.11: Workload 1: 95^{th} percentile of the average App/DB server utilisation for dynamic requests

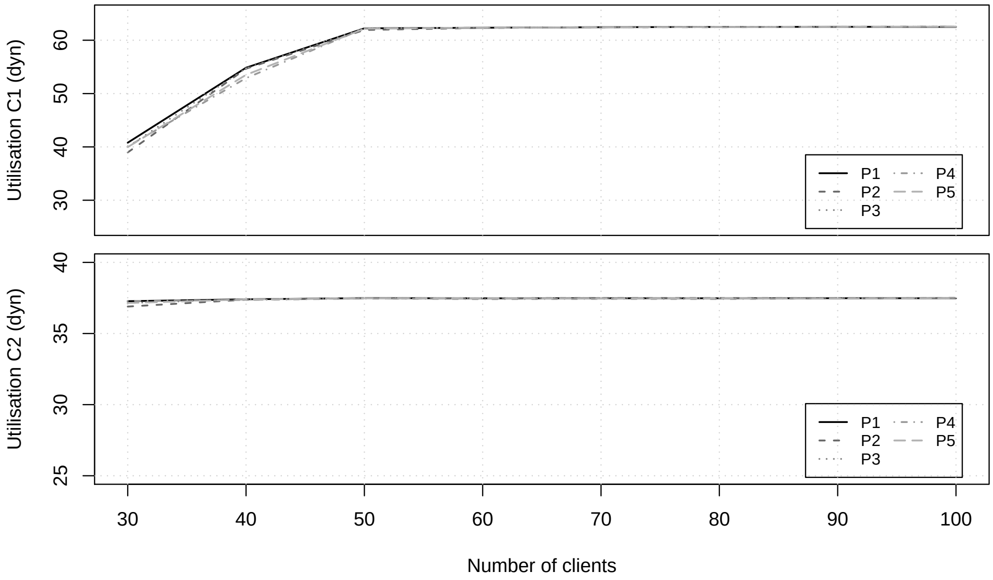


Figure 5.12: Workload 2: 95^{th} percentile of the average App/DB server utilisation for dynamic requests

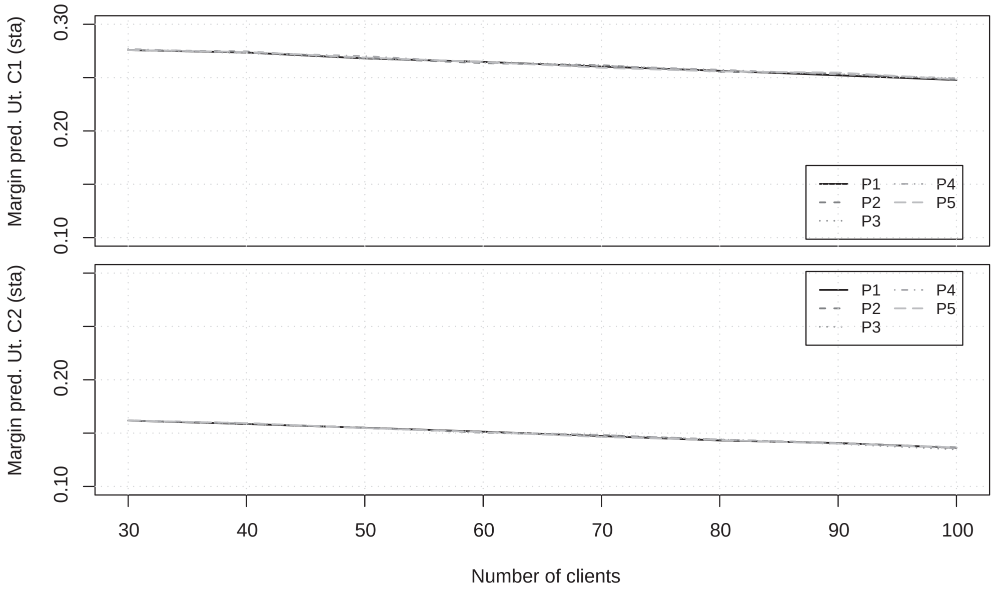


Figure 5.13: Workload 1: Mean squared error of Web server utilisation margin for static requests

It is interesting to know the margin of the utilisation left in Web and App/DB servers during the simulations. We have represented the mean squared error of the utilisation margin (described in Section 5.5) in Figure 5.13 for Web servers and *Workload 1*. We have omitted the figure of *Workload 2* as it is practically the same as Figure 5.13. The resulting curve for class-1 and class-2 requests is very similar with both workloads, despite it being a little lower for class-2 static requests as class-2 traffic has a more restrictive SLA. It can be observed that there is a slightly decreasing tendency of the utilisation margin. This is due to the increase in the number of clients which means more traffic in the Web system, and hence, a reduction in the utilisation margin for each service class.

The mean squared error of the utilisation margin of the App/DB servers is represented in Figures 5.14 and 5.15 for *Workload 1* and *Workload 2*, respectively. In this case, the resulting decreasing curve trends to zero due to the fact that the utilisation limit of each type of service is reached. We cannot see many clear differences among *Workload 1* and *Workload 2* in these figures.

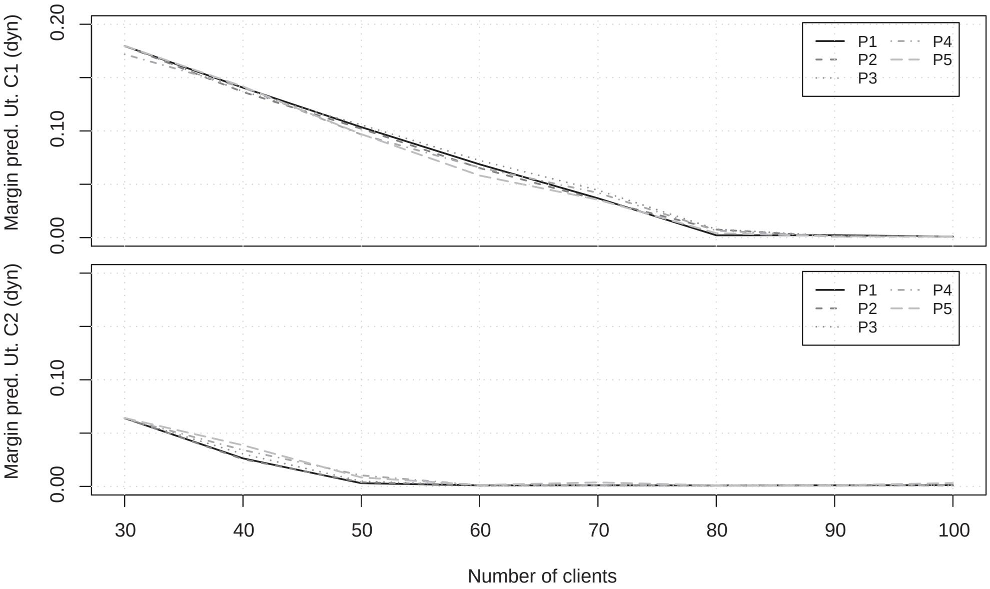


Figure 5.14: Workload 1: Mean squared error of App/DB server utilisation margin for dynamic requests

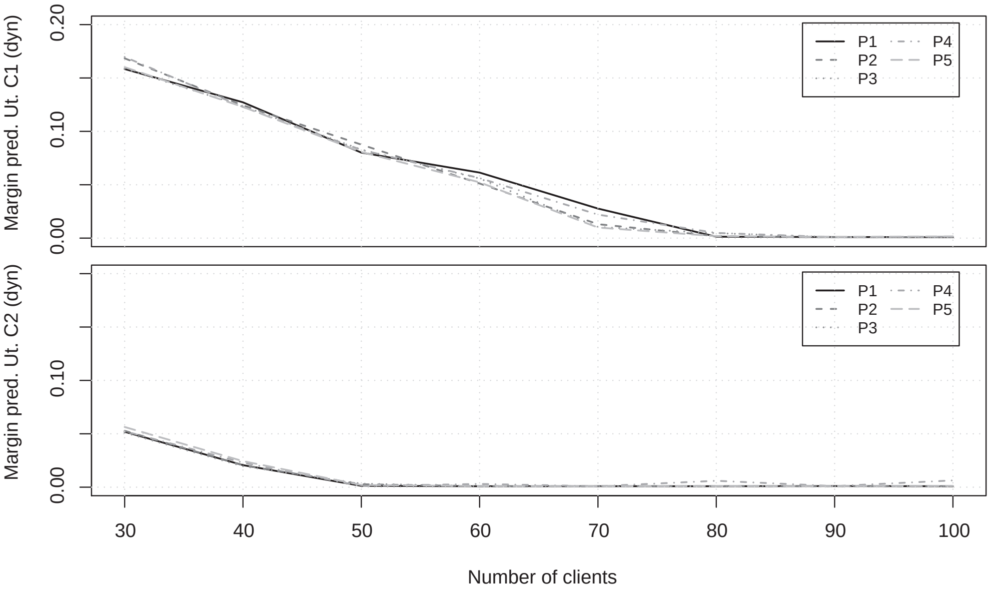


Figure 5.15: Workload 2: Mean squared error of App/DB server utilisation margin for dynamic requests

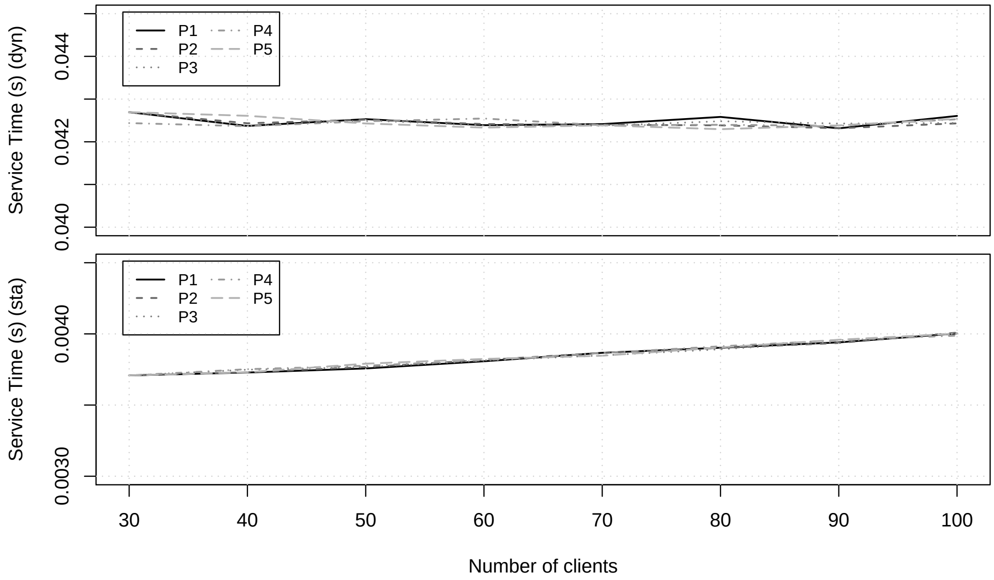


Figure 5.16: Workload 1: 95^{th} percentile of the service time in Web servers (sta) and App/DB servers (dyn)

Figure 5.16 depicts the 95^{th} percentile of the service time needed in Web and App/DB servers to serve static and dynamic requests, respectively. We have included this figure in order to have an idea of the service time values during the simulation, as it is used to predict the value of the next slot's utilisation.

We have also included the mean squared error of the utilisation prediction that compares the predicted utilisation for a slot k, $\hat{u}_{i,j,z}(k)$ and the utilisation monitored during the slot k, $u_{i,j,z}(k)$, in Figures 5.17 and 5.18 for Web servers, and Figures 5.19 and 5.20 for App/DB servers. These curves are very similar to the ones that represent the mean squared error of throughput predictions, but with a lower value because the utilisation level is always lower than 1 and the throughput values are normally greater.

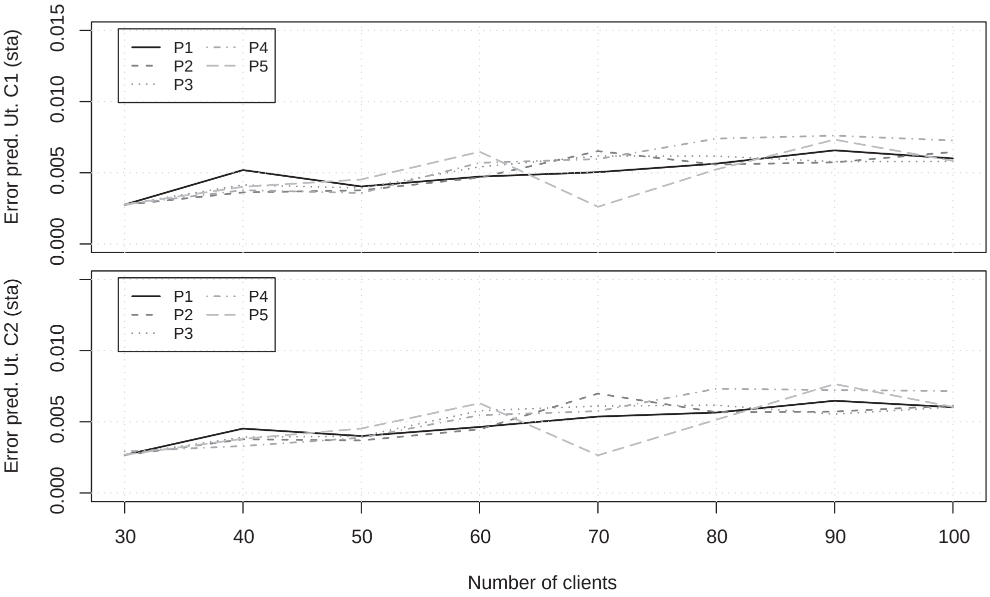


Figure 5.17: Workload 1: Mean squared error of Web server utilisation for static requests

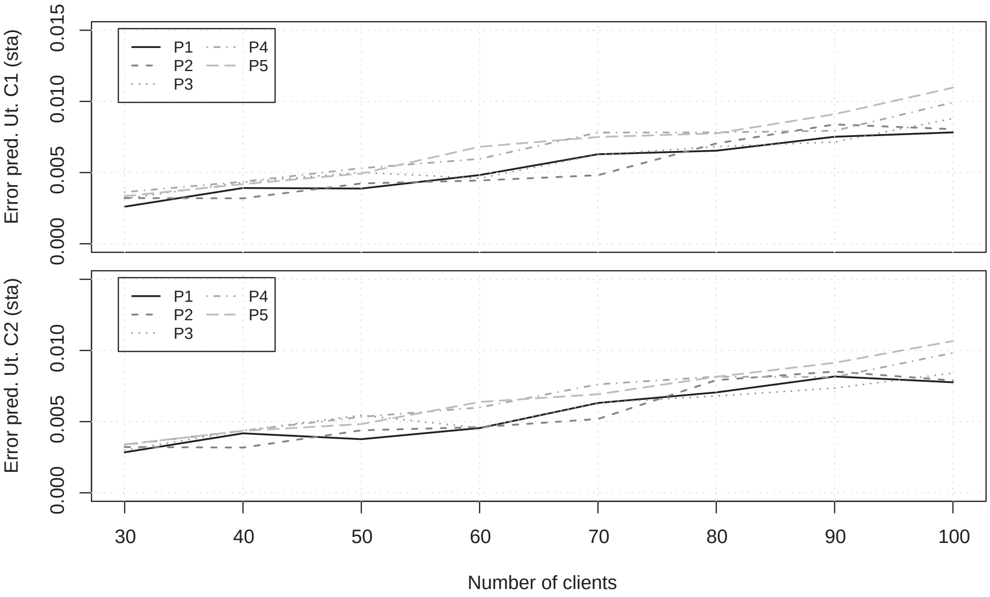


Figure 5.18: Workload 2: Mean squared error of Web server utilisation for static requests

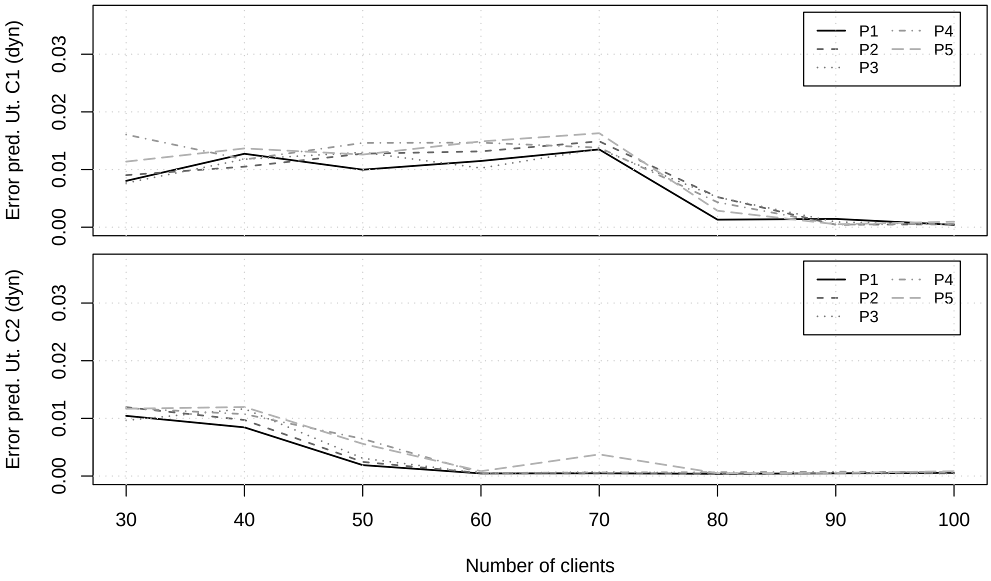


Figure 5.19: Workload 1: Mean squared error of App/DB server utilisation for dynamic requests

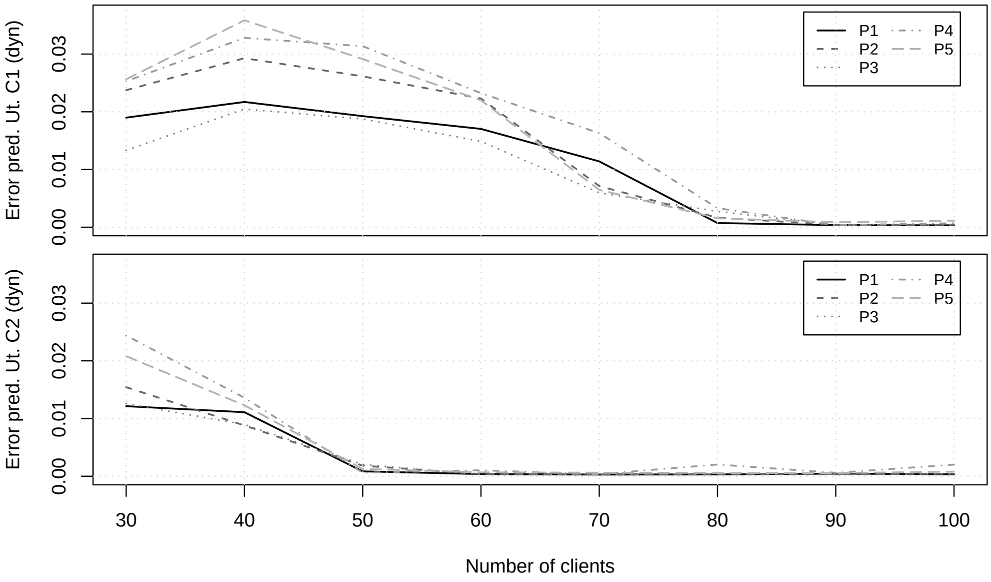


Figure 5.20: Workload 2: Mean squared error of App/DB server utilisation for dynamic requests

An important consequence of the admission control algorithm is the rejection of requests when the Web system is overloaded. Hence, let us represent the number of rejected requests during the experiments for dynamic traffic in Figures 5.21 and 5.22. We have not included the static requests as none of them is rejected in any of the simulations. As we have previously guessed when analysing the throughput prediction errors and the utilisation of the App/DB servers, with *Workload 1*, rejections start from 70 clients for class-1 requests and from 40 clients for class-2.

With *Workload 2*, rejections start earlier for class-1, that is at 50 clients, and for class-2, despite the figure not being very clear at this point, there are already some rejections with 30 clients. It is also significant to observe that the total number of rejections is greater for *Workload 2* than for *Workload 1*.

We need to analyse a metric that differentiates the behaviour of the five throughput predictors defined. So, we also consider the response time of requests in this subsection. It is represented in Figures 5.23 and 5.24 for static requests and in Figures 5.25 and 5.26 for dynamic requests.

Let us analyse first the response time of static requests for both workloads. Despite no static request being rejected, the increase in the arrival rate that represents *Workload 2* has an impact on the response time of the static requests as it can be observed in Figure 5.24, compared to the response time of these requests with *Workload 1* in Figure 5.23. We have used the same scale in both figures in order to make this difference clearer.

The response time of dynamic requests is more meaningful as the App/DB servers are congested with the increase of traffic. If we analyse the case of *Workload 1* in Figure 5.25, we can note some differences among the response time obtained by the predictors. Analysing the last case, 100 clients, we can detect that the predictors P1, P2 and P3 obtain a higher response time than predictors P4 and P5. This is also depicted in Figure 5.26, which represents the response time for *Workload 2*. We can also observe that the maximum response time for both workloads is around 2.5 seconds, that means that our algorithm achieves an extra goal, that is the limitation of the response time regardless of the amount of traffic arriving to the system. The predictor that shows a good response time and the most stable behaviour is P4, as P5 shows some variability in 70, 80, 90 and 100 clients for *Workload 1*. We can also observe that there is not any differentiation in the response times obtained by class-1 and class-2 traffic, as we do not distinguish different queues in the Web and App/DB servers in order to keep the approach simple.

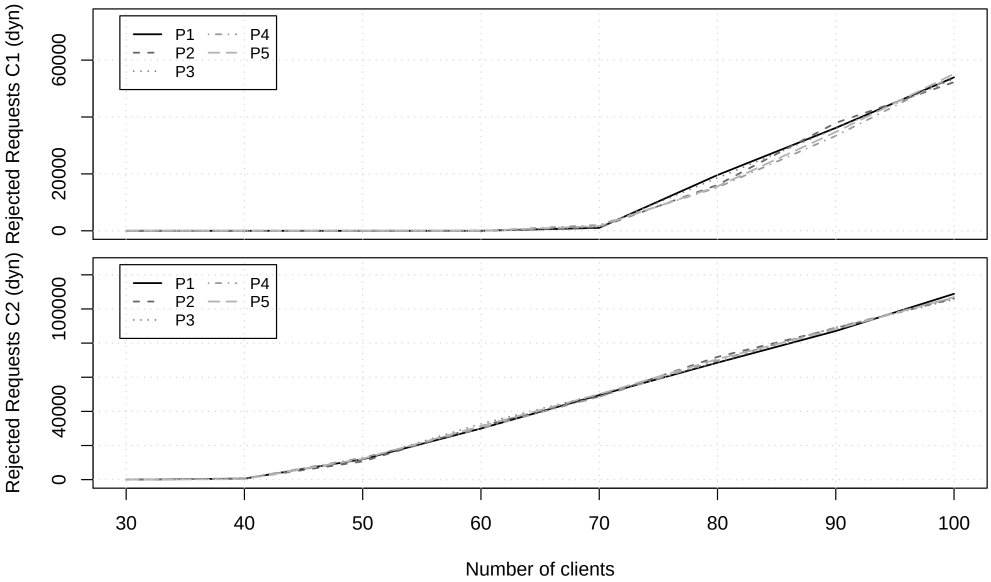


Figure 5.21: Workload 1: Number of rejected dynamic requests

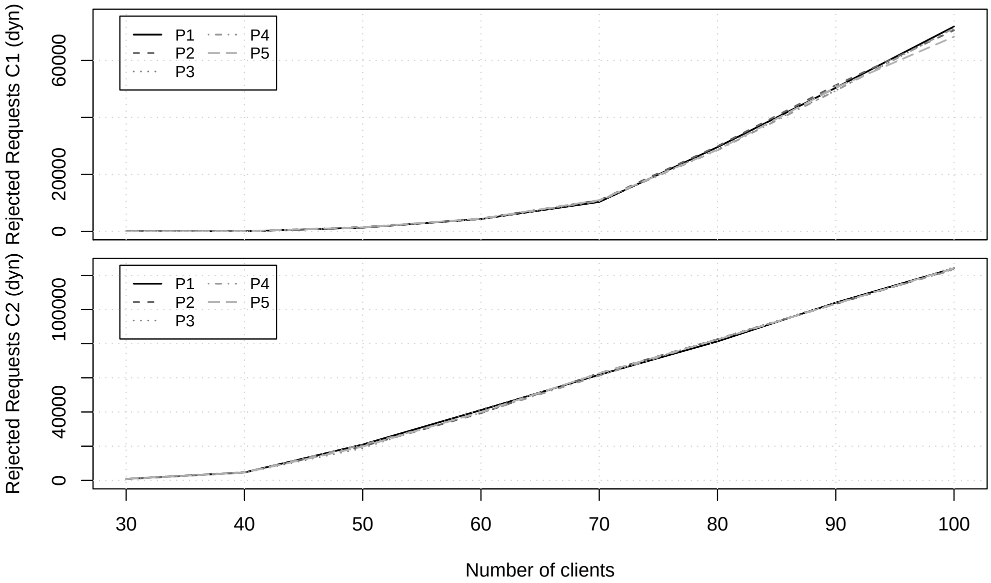


Figure 5.22: Workload 2: Number of rejected dynamic requests

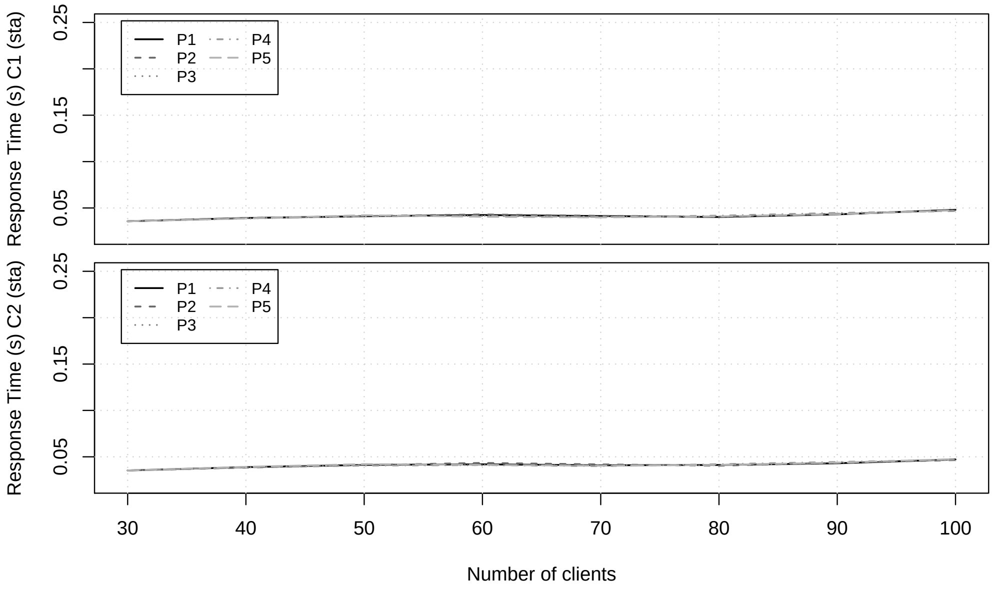


Figure 5.23: Workload 1: 95^{th} percentile of the response time for static requests

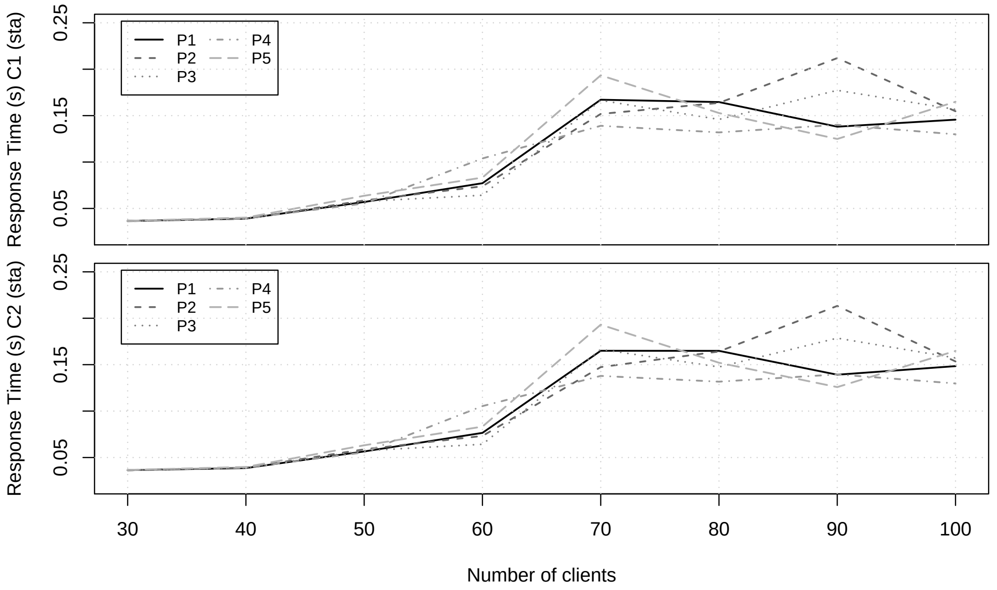


Figure 5.24: Workload 2: 95^{th} percentile of the response time for static requests

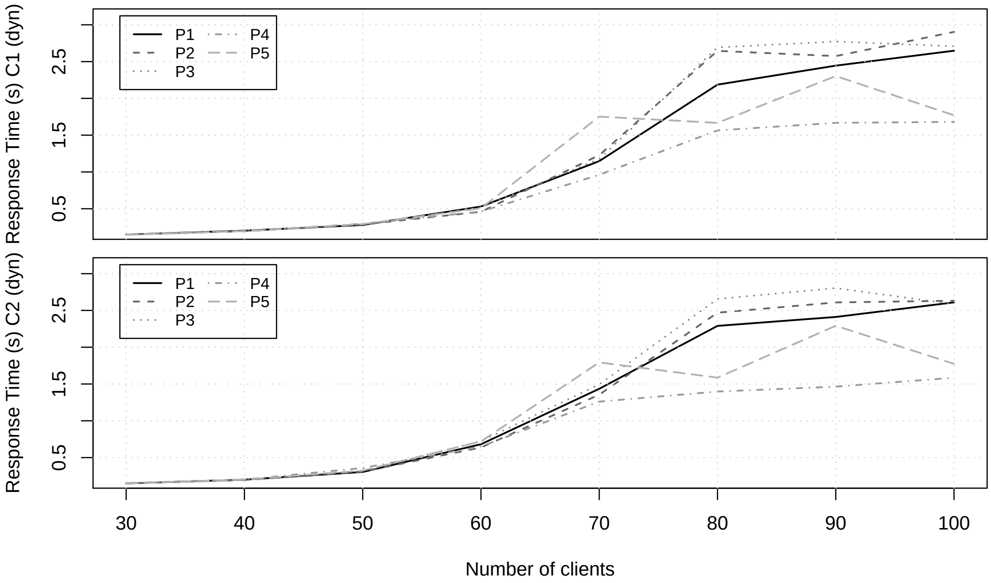


Figure 5.25: Workload 1: 95^{th} percentile of the response time for dynamic requests

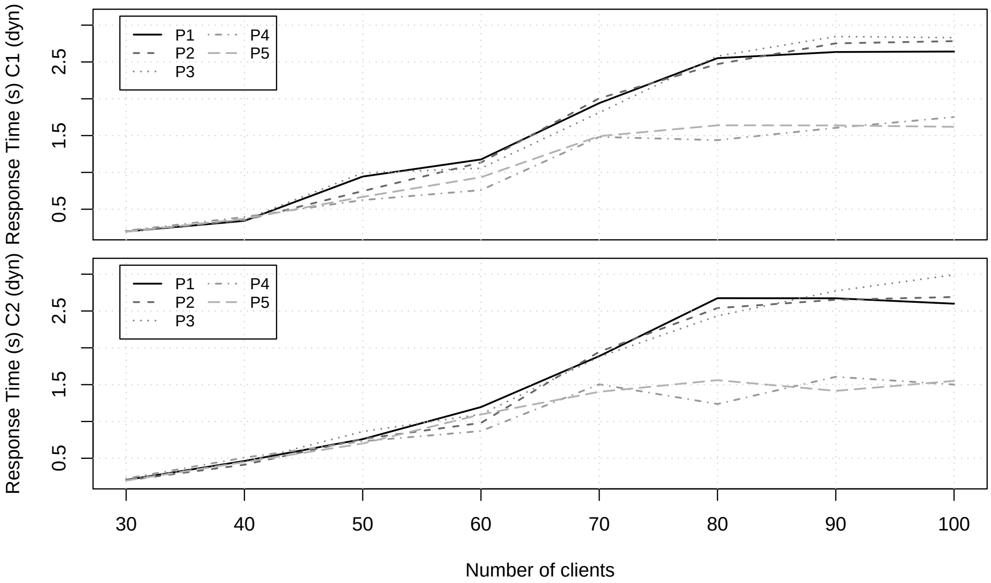


Figure 5.26: Workload 2: 95^{th} percentile of the response time for dynamic requests

The number of downloaded dynamic Web pages is represented in Figure 5.27 for *Workload 1* and Figure 5.28 for *Workload 2*. We have chosen to represent only the number dynamic Web pages because we are looking for another distinction among the predictors and the number of downloaded static Web pages would not give us that information. We can observe how, when the system receives a higher arrival rate (from 80 client to 100 clients), throughput predictors P4 and P5 download more class-1 dynamic Web pages comparatively to the rest of predictors with both workloads. We consider this more important than the fact that these predictors download less pages for 50, 60 and 70 clients, as the more traffic detected in the system, the more crucial the behaviour of the predictor is.

The response time and the number of downloaded dynamic Web pages obtained from the simulations leads us to the conclusion that P4 is the most suitable predictor for our admission control and load balancing algorithm. However, we would like to remark that the predictors P1, P2 and P3 do also obtain good performance results and that have an important advantage: they are easily obtained from the throughput of the two previous slots and that do not need a record of more previous slot throughput values as predictors P4 and P5, which are more complicated to compute.

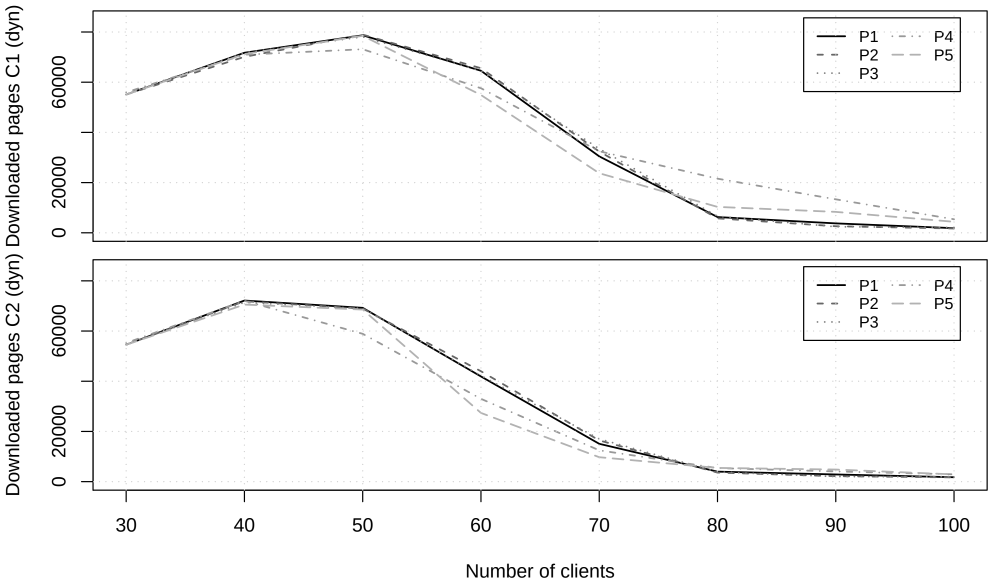


Figure 5.27: Workload 1: Number of downloaded dynamic requests

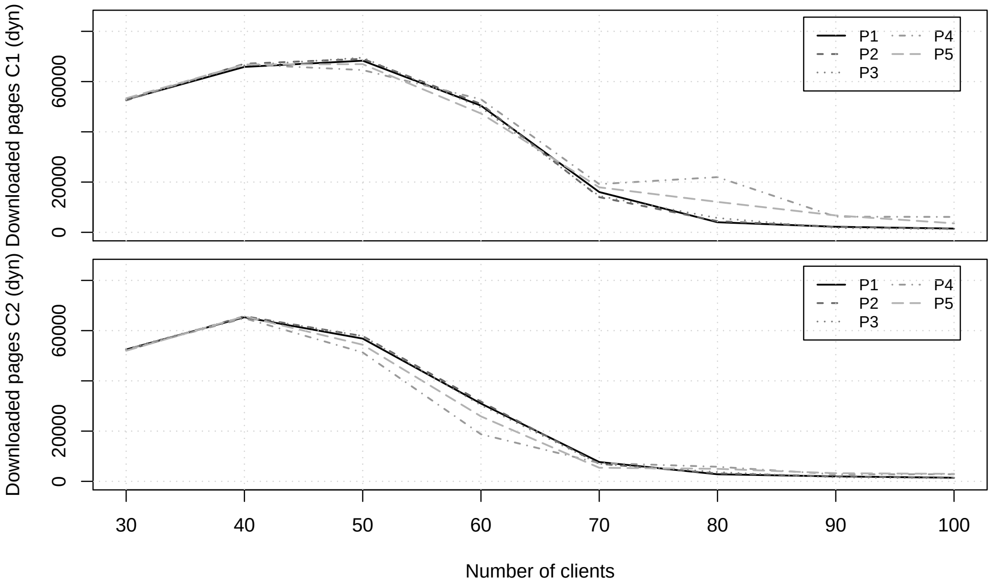


Figure 5.28: Workload 2: Number of downloaded dynamic requests

5.7.3 Adaptive Time Slot Scheduling compared to Fixed Time Slot Scheduling

We have configured the admission control and load balancing algorithm to be executed on a fixed time slot scheduling in order to compare its results with our adaptive time slot scheduling. The predictor chosen for these simulations is P3. We have redrawn some of previous figures, including the results obtained when invoking the algorithm periodically (named as "P3_per" in the figures). The workload chosen for this comparison is *Workload 2*, described in the previous subsection.

The first figures we have included in this comparison are the mean square error of throughput predictions for both static and dynamic traffic. It can be observed in Figure 5.29 the error of static throughput predictions and in Figure 5.30 the error of dynamic predictions. Very similar results are obtained for the adaptive time slot scheduling and the fixed time slot scheduling, so the prediction errors do not help in differentiating the scheduling policy.

Let us analyse therefore other metrics such as the number of rejected dynamic requests and the utilisation level of the App/DB servers. We omit the utilisation level of the Web servers because, as we have seen in the previous subsection, they are linear and do not add any extra information.

Figure 5.31 shows the number of rejected requests that ask for dynamic content. Once again, we cannot observe either a differentiation among the predictors, or in the scheduling.

The 95^{th} percentile of the App/DB server utilisation is represented in Figure 5.32. Here we observe that the utilisation level of the App/DB servers is lower for P3_per in the first points of the x-axis of the graph. In the case of class-1 traffic, the servers seem to be less loaded for 30, 40 and 50 clients with P3_per. The case of 30 clients also reaches a lower utilisation level for class-2 traffic. However, if we analyse the P3_per utilisation level of class-2 traffic after 40 clients, we can also observe that it is slightly greater that the rest of the simulations. In fact, this indicates to us that the fixed time slot scheduling introduces some errors in the utilisation level reached for each traffic class. That also means that the algorithm is less accurate in its reservations and that the SLA is less guaranteed.

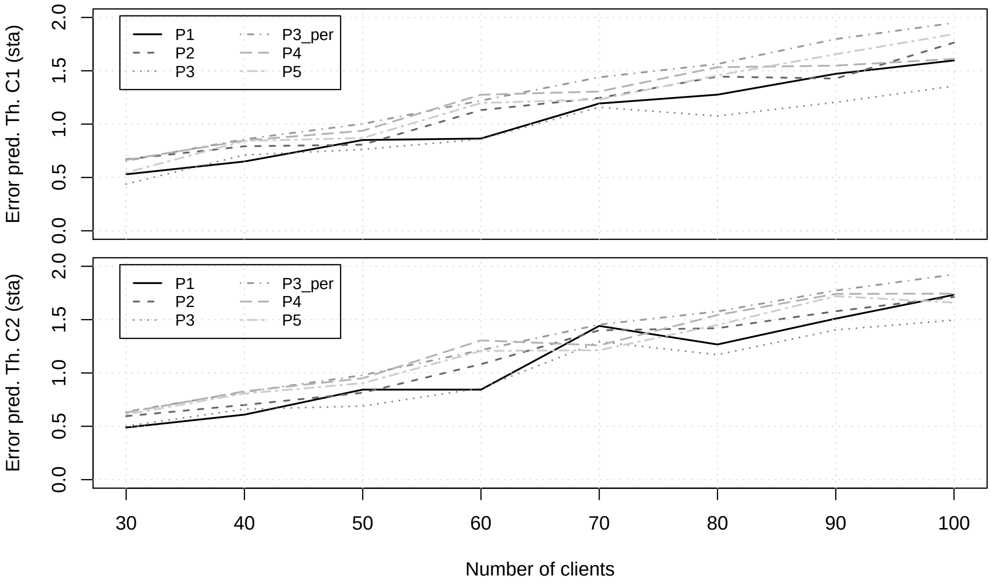


Figure 5.29: Mean squared error of static requests' throughput predictions

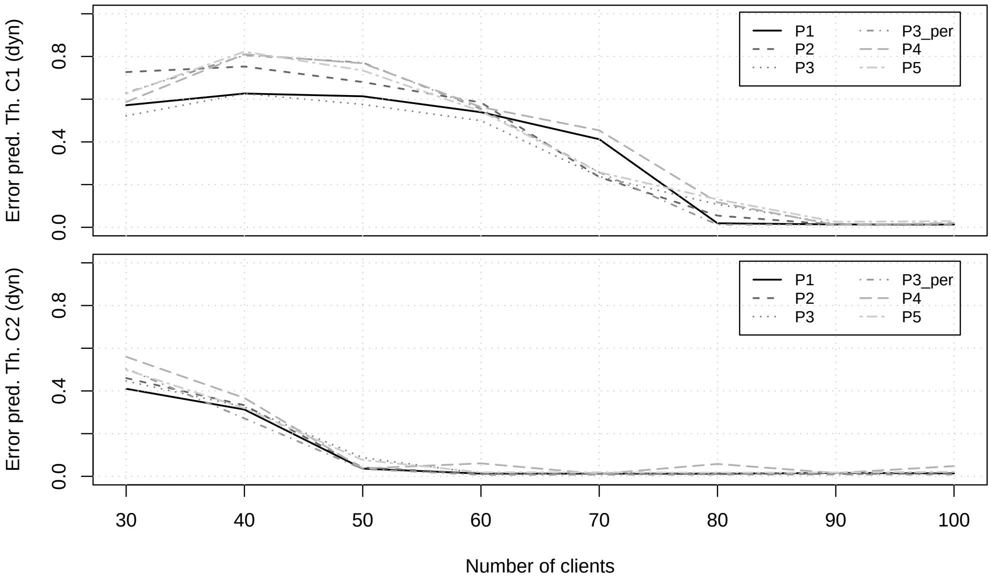


Figure 5.30: Mean squared error of dynamic requests' throughput predictions

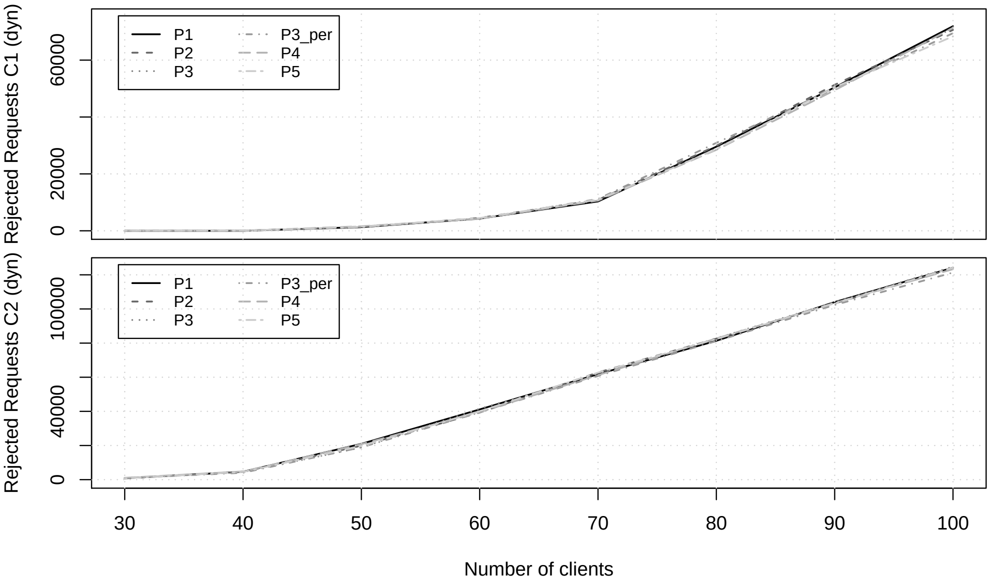


Figure 5.31: Number of rejected dynamic requests

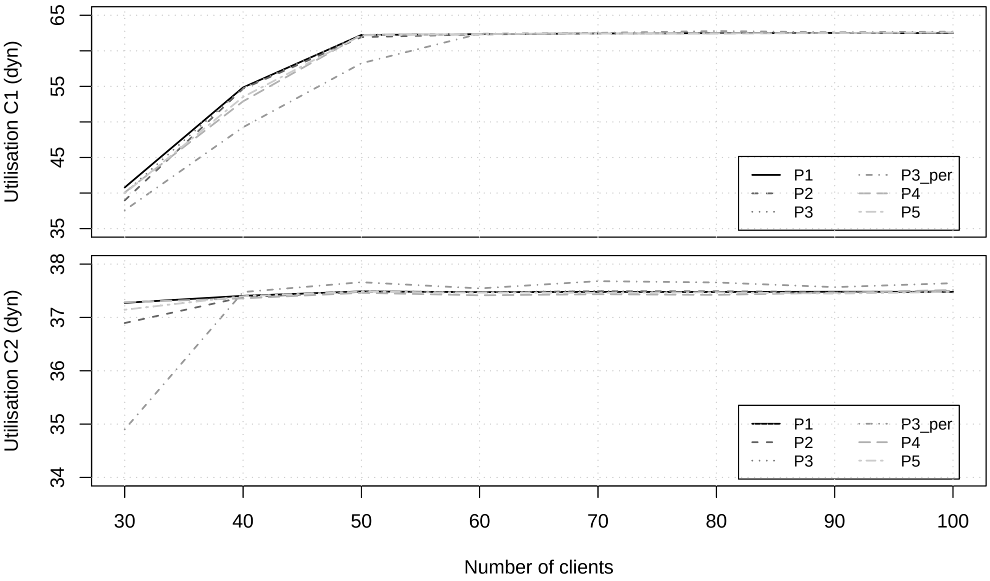


Figure 5.32: 95^{th} percentile of the App/DB server utilisation for dynamic requests

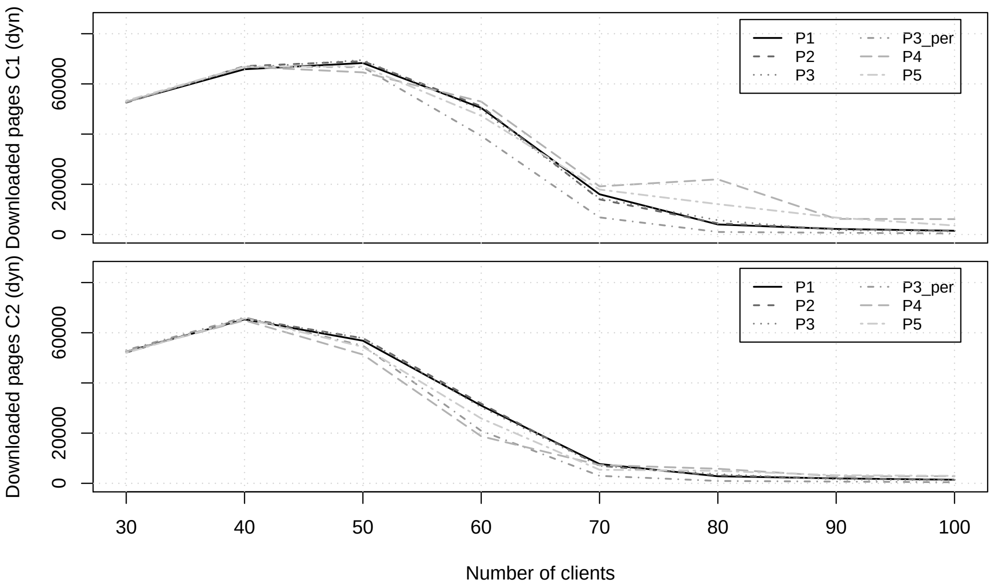


Figure 5.33: Number of downloaded dynamic requests

The response time and the number of downloaded requests gave us an idea of the predictor that suits our algorithm best in the previous subsection. So, let us redraw these figures again in order to know the behaviour of the P3 predictor under a periodical invocation scheme.

Figure 5.33 shows the number of downloaded dynamic requests. It denotes that P3_per downloads less class-1 requests than the rest of the simulation results. Also we can observe this same behaviour for class-2 traffic. Despite P3_per utilisation level for class-2 requests being greater than the rest of the simulations (as we have just seen in Figure 5.32), it downloads less class-2 requests.

Finally, Figures 5.34 and 5.35 show the response time obtained by static and dynamic requests, respectively. The results obtained by the static requests do not give us much information, but the response time for dynamic request do. We can observe in Figure 5.35 how P3_per increases its response time for most of the number of clients above the rest of the curves.

This confirms the fact that the simulation of the algorithm following a periodical invocation is outperformed by the adaptive invocation.

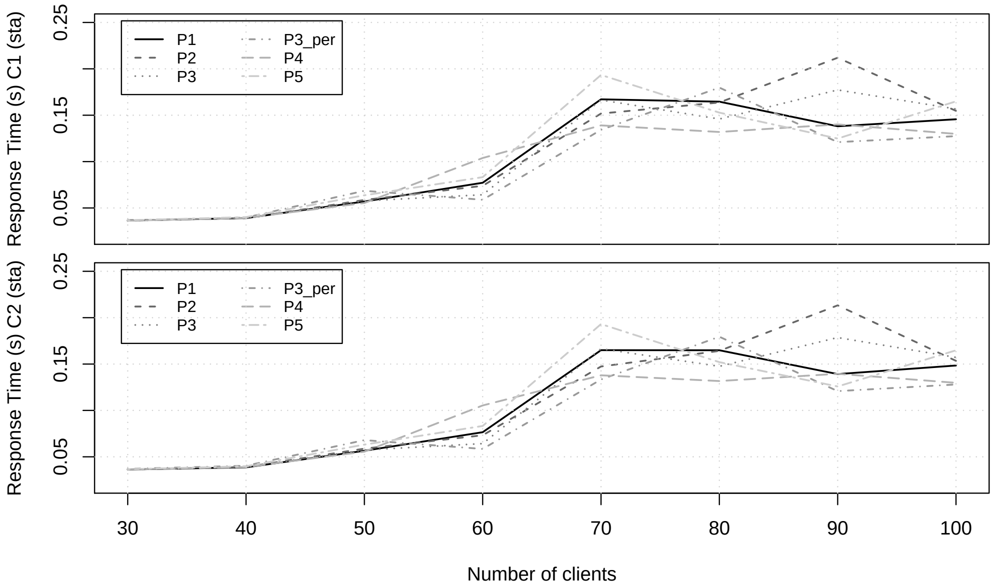


Figure 5.34: 95^{th} percentile of the response time for static requests

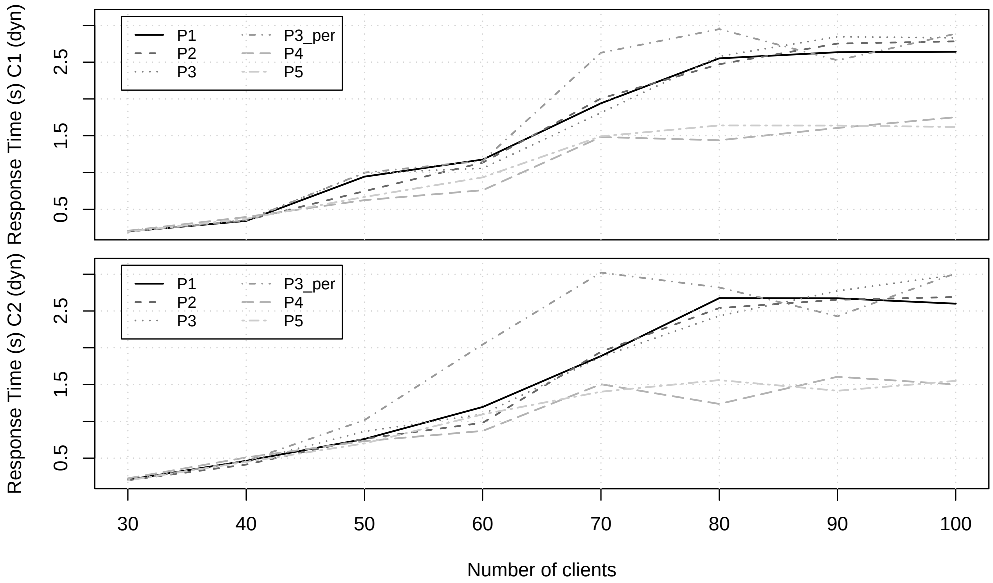


Figure 5.35: 95^{th} percentile of the response time for dynamic requests

5.7.4 Comparison to IQRD

In Chapters 2 and 3, we reviewed the Web load balancing proposals and the admission control mechanisms that exist in literature, and finally we have found a candidate proposal to be compared with our algorithm. It has been difficult to find because we needed the alternative solution to be QoS-aware and include an admission control and a content-aware load balancing strategy.

We decided to compare our algorithm with the IQRD, the solution described by Sharifian *et al.* in [123]. An important reason we have considered to select IQRD is because it divides the time into small intervals in order to avoid modelling errors. They consider a periodical algorithm invocation, and at the same time these periods (or slots) are divided into smaller intervals in order to obtain monitoring information from the Web servers. This is a completely different philosophy from ours and the results of this comparison may be quite interesting and lead us to make some conclusions about the possible reduction of overhead when invoking an algorithm in a Web environment.

The IQRD solution defines the SLA of the different services by the response time. Several Web servers are supposed to execute in a node in order to differentiate among the services offered. Each of the Web servers is modelled by a $M/P/1$ queueing network. Hence, the inter-arrival rate the IQRD model uses follows an exponential distribution. The response time guaranteed for each type of service permits it to obtain the maximum arrival rate that is going to be accepted during the next period. In order to obtain this value, the mean and the second momentum of the service time are received from the Web servers. These metrics are monitored 100 times per period (or slot). The maximum arrival rate permits it to know the maximum number of requests of each service type which is going to be accepted during the next period.

For a period, the maximum number of requests decreases by one in a Web server when a request of that class is accepted, and increases again when the request has been served. This would add an extra overhead in a one-way architecture as the server nodes should inform the distributor that a request has been served. The admission control starts rejecting requests when all the Web servers have the maximum number of requests equal to zero.

In our implementation of OPNET Modeler, we have considered that only one Web server is executing in each Web and App/DB node. Hence, in order to simulate both algorithms under the same conditions, we have defined two different classes of service, c_1 and c_2, that in the case of IQRD have the same response time requirement, and in our case, have the same SLA value. Requests from each class of service can be either static or dynamic as we have previously described in Subsection 5.3.2.

In order to represent our algorithm, we have chosen to simulate the predictor P4 with the SLA values $c_1 = c_2 = 0.5$. The workload used in both sets of simulations is the same as the *Workload 2*, that is described in Subsection 5.7.2.

We have represented the number of downloaded dynamic pages in Figure 5.36. It can be observed that both curves are very similar, although IQRD downloads more class-1 requests for 60 and 70 clients than our algorithm, and our algorithm outperforms IQRD for 80 and 90 clients. However, both of them essentially download a similar number dynamic requests.

If we compare the plot of number of downloaded requests with Figure 5.37, that shows the response time of dynamic requests for both algorithms, we can observe the two curves are very similar as well. The response time obtained by both classes is always below 1.6 seconds, despite IQRD achieving a better response time in the most extreme cases, that is with 80, 90 and 100 clients.

Figure 5.38 depicts the number of dynamic rejected requests. In this figure it is clear that IQRD rejects more requests for most of the number of clients than our algorithm.

Therefore, we can conclude in this subsection that our algorithm obtains good performance results in comparison to another admission control and content-aware load balancing algorithm that considers QoS, and is named IQRD. We have compared these two algorithm under the same circumstances and we have observed that both of them work well, despite our algorithm rejecting less requests considering the same workload.

Several comments should be included as conclusions. In our aim to obtain low overhead, we do not consider a periodic invocation time in our algorithm as IQRD does. Indeed, IQRD algorithm requires always fresh information from the servers, and subdivides the intervals 100 times to get information from them. This will increase the overhead of the solution. But, there is also another detail in the IQRD proposal that will increase the overhead of the system, and it is the fact that the algorithm needs to control the number of requests that can still be attended by the server: when they decrease, and more importantly, increase. When the service of a request is finished, the IQRD proposal requires an extra feedback from the Web servers as they have to indicate to the distributor module that the counter should be increased by one. Hence, we prove with these results that our algorithm obtains as good performance as that obtained by an algorithm that introduces more overhead than

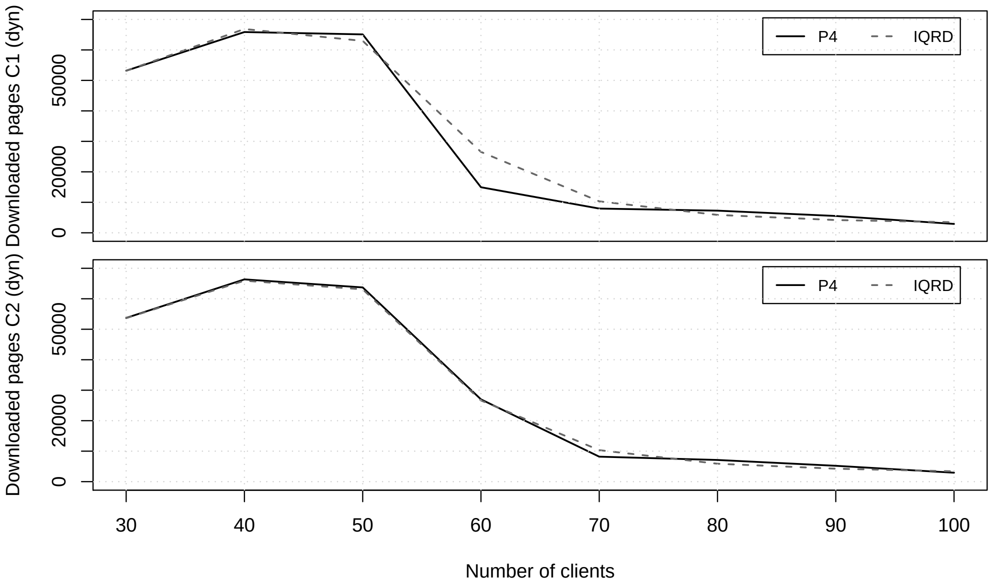


Figure 5.36: Number of downloaded dynamic requests

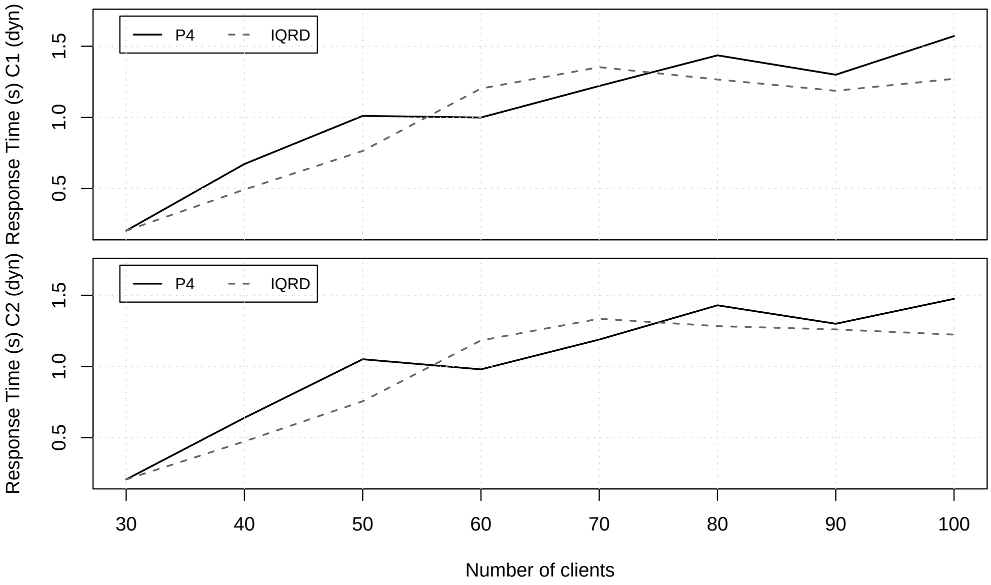


Figure 5.37: 95^{th} percentile of the response time for dynamic requests

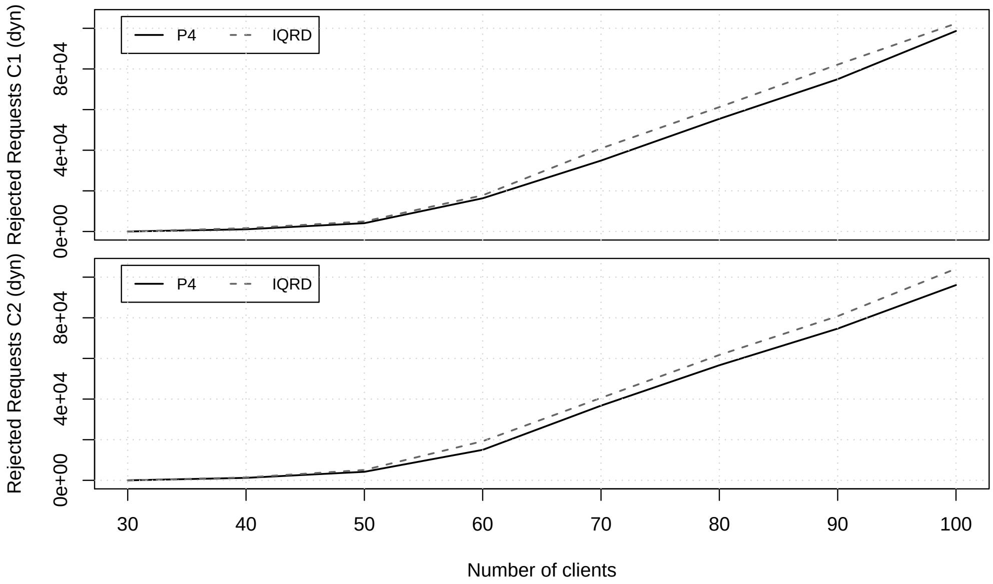


Figure 5.38: Number of rejected dynamic requests

ours in the system.

5.8 Summary

We introduce a low overhead admission control and load balancing algorithm that bases its decisions on the values obtained by a throughput predictor. The invocation times of the algorithm are adaptively scheduled depending on the burstiness detected in the system in order to reduce the overhead of the algorithm. Five throughput predictors are introduced in this work and their behaviour in the admission control and load balancing algorithm is compared under a simulation scenario in OPNET Modeler.

The resource allocation algorithm adaptively distributes the utilisation of the servers among the different classes of requests. The results show that the algorithm guarantees the service specified in the SLA with all the throughput predictors. However, the results also show differences in the response times of the requests that ask for dynamic content as the bottleneck of the system starts to be overloaded. These differences plus the number of downloaded dynamic requests lead us to the conclusion that the predictor that better suits to our algorithm is based on LMS. An extra goal is acquired by this algorithm: that is the limitation in the response time of the requests when it is congested. This means that service is guaranteed despite the arrival rate that is reaching the Web system.

We have also compared two classical load balancing policies to be included in the algorithm: RR and LC detecting that if not many servers are active for a particular type of service, then the RR policy balances the incoming requests in the cluster better because it sends the same number of requests to all the active servers. The LC policy in this case sends all the requests to the server with the least number connections until the number of connections with the rest of *active* servers are equal, which may quickly result in an overload situation.

In this chapter, we have also provided results that show the benefits of an adaptive time slot scheduling compared to a fixed time slot one.

The comparison of the results obtained by our algorithm and IQRD shows us that our algorithm works reasonably well and its performance is very similar to the performance of IQRD while needing less overhead.

Part III

Conclusions, Appendix and References

Chapter 6

Conclusions and Open problems

In this last chapter we sum up the conclusions we have obtained in this work and its future trends.

6.1 Conclusions

The main goal of this dissertation is the design a low-overhead adaptive admission control and content-aware load balancer algorithm for a Web system.

We have organised this dissertation in two main parts that are the *background* and the *contributions*. Hence, we also structure the conclusions in the same way.

Background We have included two chapters in the *background* which introduce the concepts regarding Web load balancing and, burstiness and admission control in a Web system:

- Chapter 2 introduces a thorough survey of the Web load balancing solutions separating the network architecture and distribution policy. A classification is introduced by distinguishing the OSI protocol stack layer the load balancing mechanism is based on, concluding that content-aware mechanisms are more useful than content-blind mechanisms because of traffic differentiation. However, some content-aware solutions include content-blind mechanisms and leave the distribution task to be done by the Web servers in order to avoid the drawbacks that a layer-7 front end introduce in the Web system.

  The recap of the distribution policies shows the interest by the research community for locality-aware solutions during the first years of the decade 2000 as they try to improve the cache performance. Hence, these proposals normally only take into account requests that ask for static Web content. More recent proposals consider dynamic Web content, but it is a field that continues being researched as the service times of the scripts that generate the dynamic content are not easily predictable. We situate our work in this context.

- Chapter 3 reviews recent proposals that avoid a congestion situation in the Web system such as burstiness detection and admission control solutions. Burstiness detection is introduced in several areas of research, although we are mainly interested in areas related to network traffic. Some of the proposals described in this survey are considered in Chapter 4 in order to compare them with our burstiness factor.

  Admission control proposals are also studied and classified. We observe that they consider either a periodical or non-periodical invocation frequency. The non-periodical invocation is normally activated by the arrival of a new request or session to the system. We consider that the arrival rate has to be able to adaptively alter the invocation frequency in order to avoid a possible congestion situation in the Web system.

Contributions This part also includes two chapters:

- We have defined and studied six different burstiness factors in Chapter 4. Based on the burstiness factor, an adaptive time slot scheduling is described. It is used to set the monitoring frequency of the metrics used in the admission control and load balancing algorithm, and also to adaptively set the slot duration based on the bursty arrivals detected at the system.

  Among the burstiness factors studied in Chapter 4 we have found that, on one hand, BF1, BF5 and BF6 are very conservative and do not vary their values significantly with the changes in the arrival rate. On the other hand, the factors that include a penalisation when detecting successive bursty slots (BF3 and BF4) reach the maximum value easily, hence, further increases in the arrival rate are not detected. We select BF2 as the burstiness factor to be included in our algorithm because permits it to be aware of an increase in the arrival rate independently of the actual workload in the system.

- Chapter 5 describes the admission control and load balancing algorithm we have developed, which is based on the adaptive time slot scheduling defined in Chapter 4. The adaptive slotted time described in Chapter 4 is also used to invoke the algorithm. Therefore, our algorithm takes into account the burstiness factors for its own invocation times. The admission control mechanism adaptively distributes the utilisation of the servers among the different classes of request based on the throughput prediction and considering QoS.

  The algorithm distributes the servers' utilisation among the classes of service guaranteeing the SLA requirements. We have compared five throughput predictors in Chapter 5 and conclude that the predictor that better suits our algorithm is based on LMS as it is the one with the least response time and the most stable behaviour. An extra goal is acquired by our algorithm in the limitation in the response time of the requests when it is congested. This means that service is guaranteed despite the arrival rate that is reaching the Web system.

  Our load balancing strategy is based on a classical policy. Hence, we have compared two policies: RR and LC detecting that if not many servers are active for a particular type of service, then the RR policy balances the incoming requests in the cluster better than LC.

  Some results are also obtained of a fixed time and our adaptive time slot scheduling observing that the second outperforms the first in terms of CPU utilisation, number of downloaded pages and response time. We have observed that the fixed time slot scheduling does not reserve the utilisation as accurately as the adaptive one does.

  We have compared our algorithm to another QoS-aware admission control and content-aware load balancing proposal, that is IQRD, in order to know their contrasted performance and found that our algorithm manages to get the same performance than IQRD with less overhead. Indeed, it rejects less requests while getting more or less the same response time. Hence, we can conclude that our algorithm guarantees the SLA contracted and comparatively means less overhead for the Web system.

6.2 Future Work

There are some open problems in our work that can be further researched.

- On the design of a burstiness factor that includes a penalisation in Section 4.3, the decision of the penalty amount to be included in it has to be more studied. Especially in the case of detecting several consecutive bursty slots. The factor has to be suitable for all possible arrival rates a Web system may expect and has to follow arrival rate variations accurately in the range of values defined for it.

- We have developed the burstiness factor to be aware of the arrival rate considering no differentiation among the different services the incoming requests ask for. Hence, a further study would be to include traffic differentiation in the burstiness factor. It may improve the performance of the Web system as all types of service may not be demanded with the same intensity in the Web system.

- Considering the admission control algorithm described in Section 5.5, the resource's reservation in the Web servers is done regardless of the amount of service demanded. Hence, it may happen that, when a low priority service class reaches its maximum utilisation in all the servers, the requests asking for this service are rejected. This can lead to an under-utilised reserved utilisation if no more priority requests arrive to the system during that slot. This problem may be solved by defining a probability index based on a record that stores the percentage index of arrival rate asking for each class of service during some previous slots, that would estimate the chance of not receiving more priority requests, and allow an increase in the utilisation margin of the lower priority requests. However, this point would need further study. We leave this possible modification of the resource allocation strategy as a future work because our aim in this work was to check if the admission control algorithm effectively reserves the utilisation that would guarantee the SLA requirements.

- We have tested our algorithm in a simulation scenario based on TCP hand-off, which has the drawback of not being scalable [14]. The reason for choosing TCP hand-off for our simulation model is described in Appendix A. But, in case our algorithm is implemented in a real prototype, an alternative more scalable load balancing architecture has to be chosen. Hence, the modifications that would need to be done in our algorithm in this case need also to be studied.

- Our algorithm is developed to be included in a locally distributed Web system, but could also form part of a wider geographically distributed load balancing architecture.

This possibility has to be further researched.

Appendix A

Implementation in OPNET Modeler

In this appendix we briefly describe how we have implemented our model in the simulation tool OPNET Modeler and the modifications we have done to some of its standard models in order to adapt them to our test bed configuration. Some parts of this appendix appear in [64] and in [65].

A.1 Introduction

The tool OPNET Modeler 11.0 [3] has been used to implement the simulation model. The architecture of the complete OPNET simulation model is represented in Figure A.1. The Web cluster consists of up to 20 servers organised in a 2-tier architecture that are connected to a Web switch (1 in the figure). Static HTTP requests are attended by the Web servers (2), while dynamic requests are addressed to the App/DB servers (3) from the Web servers in order to query the information requested.

The clients (4) are organised in a switched Local Area Network (LAN), that is configured with the number of clients needed for each simulation.

The inclusion of the OPNET standard Application (5) and Profile (6) modules permits us to configure the different profiles executed by the clients and the applications included in these profiles. The module Tasks (7) allows us to configure the custom applications needed to simulate the requests that access to the App/DB servers to retrieve the dynamic information. In order to specify the parameters of the jobs executed in the servers, we have

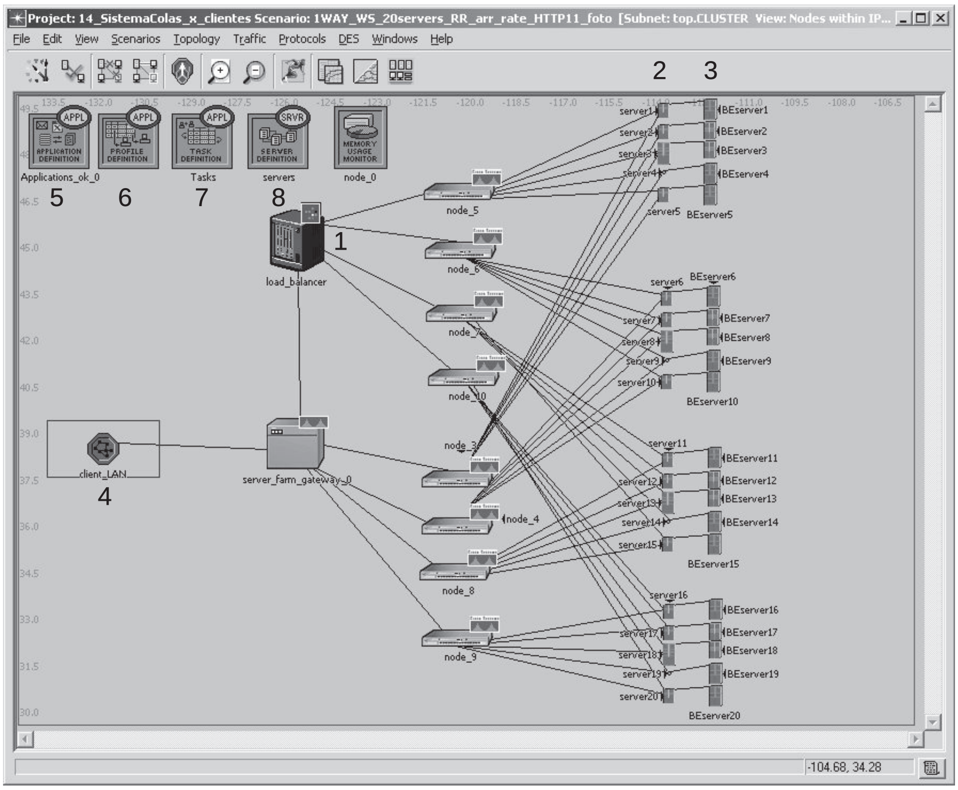


Figure A.1: Architecture of the simulation model

used the Server definition module (8).

We have modified the code of some of the standard OPNET models in order to get a precise and realistic scenario. We chose to implement a layer-7 Web switch that distributes the incoming HTTP requests based on the TCP hand-off mechanism despite the drawbacks we commented on in Chapter 2 for several reasons:

1. Our algorithm is independent of the mechanism used in the simulations. Other underlying content-aware load balancing architectures can be used to implement the algorithm. This change would not affect to the general behaviour of our algorithm, despite requiring some additional modifications in the implementation.

2. It is the easiest way to implement a content-aware load balancing architecture in OPNET Modeler because of the structure of its models.

3. The most important: after the implementation, we checked if the utilisation of the Web switch may affect the results of the simulations due to scalability problems, and

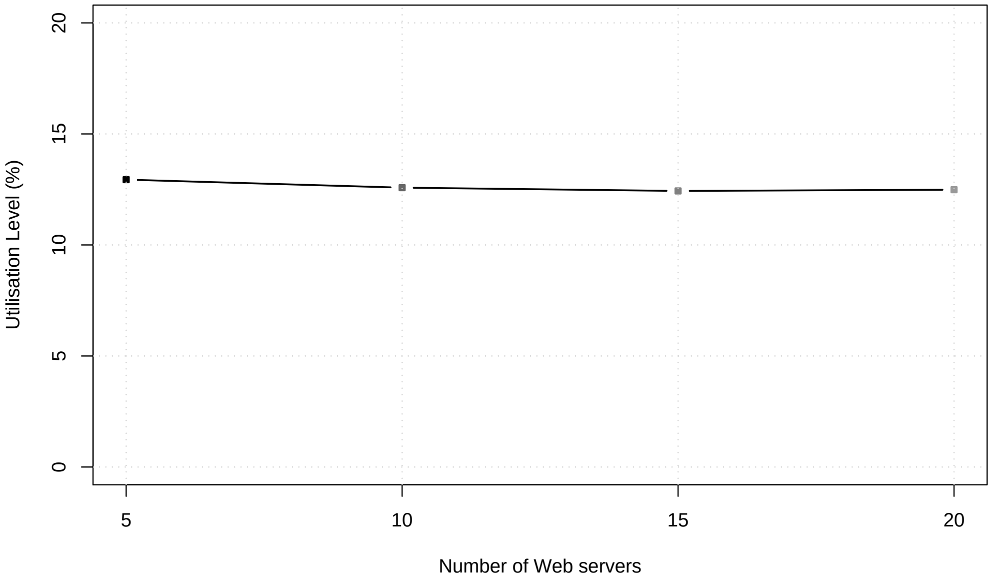


Figure A.2: 95^{th} percentile of the load balancer CPU utilisation

we found that the Web switch utilisation is stable when increasing the number of servers as shown in Figure A.2.

Hence, we decided to use the TCP hand-off mechanism for all the simulations.

Among all the modifications done to the OPNET standard models. we have chosen to describe here the most significative ones, which are the design of the process that models the Web switch and the implementation of the TCP hand-off.

A.2 Web switch Implementation

The Web switch design is based on a Layer-3 load balancer model that is already implemented in OPNET models: and that is named `gna_load_balancer`. The load balancer process has been modified to de-encapsulate the incoming data frames in order to access to the application layer information. Once the load balancer knows the application a packet belongs to, it sends the packet to the corresponding Web server following the decisions taken during the last execution of the algorithm. Hence, the load balancing is done at the application level. The process that balances the load in the Web switch is represented in

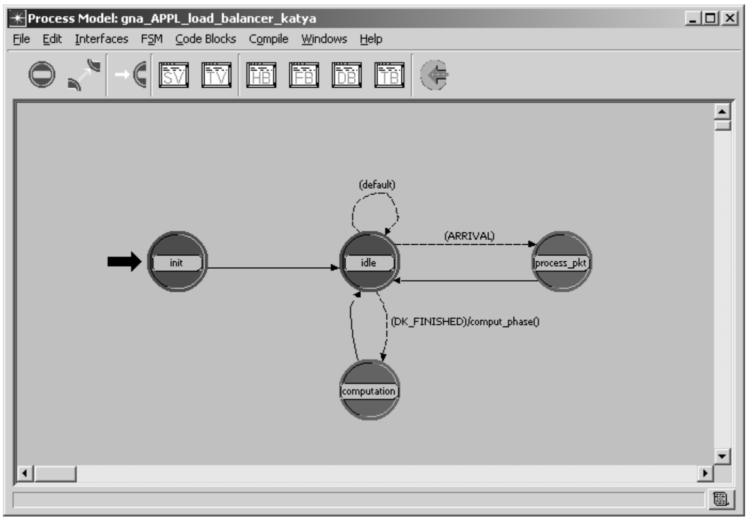

Process Model: gna_APPL_load_balancer_katya
File Edit Interfaces FSM Code Blocks Compile Windows Help
SV TV HB FB DB TB
(default)
(ARRIVAL)
init
idle
process_pkt
(DK_FINISHED)/comput_phase()
computation

Figure A.3: Web switch process

Figure A.3.

Let us describe more deeply how this process works. We have defined a timer that controls the durations of the slots. The duration of the next slot is decided during the actual one, hence, the timer is set each time the algorithm is invoked.

The Web switch process consists of four states:

- `init`: In this state, the variables and functions used in the entire process are initialised.

- `idle`: The machine enters the idle state and waits for an incoming event. The event can be either an incoming packet or when the clock timer that executes the algorithm expires.

- `process_pkt`: This state is entered when a packet is received in the load balancer. Therefore, its role is to analyse the packet in order to know to which application does it belong to and, in the case that it is a valid application for load balancing, then the process calls a function that will select the appropriate Web server to attend this new request.

- `computation`: The code of the implementation of our algorithm is included in this state. It is executed each time the clock timer expires.

This process interacts with the TCP and application layer models. Some of them have also been modified. We describe the implementation of the TCP hand-off in the next section.

A.3 TCP hand-off Implementation

We have implemented the TCP hand-off mechanism in the TCP protocol in OPNET Modeler following the directions in [77, 107].

To do this we have modified the finite state machine of the TCP protocol by adding two additional states that are marked by a square in Figure A.4, that are the `HAND_OFF` and the `HO_FORW` state.

Once the connection among the client and the load balancer is established, the connection parameters have to be transmitted from the load balancer to the Web server that is going to attend this client Web request. The hand-off mechanism begins by sending a SYN to the selected server that includes the client IP address, the Initial Send Sequence (ISS) number of the connection and the Initial Receive Sequence (IRS) number. The packet also includes the original HTTP request from the client.

After this step the load balancer's TCP process waits in the `HAND_OFF` state for a response that has to come from the server. This response should include the SYN flag activated and then, the load balancer responds with an ACK to complete the three-way handshake. Once the second connection is established among the load balancer and the Web server, the TCP process of the first one is in the `HO_FORW` state while the Web server's TCP process is in the `ESTAB` state.

The server node changes the value of the remote IP in the connection parameters in order to use the client's IP address rather than the load balancer's as the response packets are going to be sent directly to the client. Once a connection is handed-off to a Web server, incoming packets on that connection are forwarded to the Web server that is going to attend them and the corresponding responses are sent directly to the client without going through the load balancer.

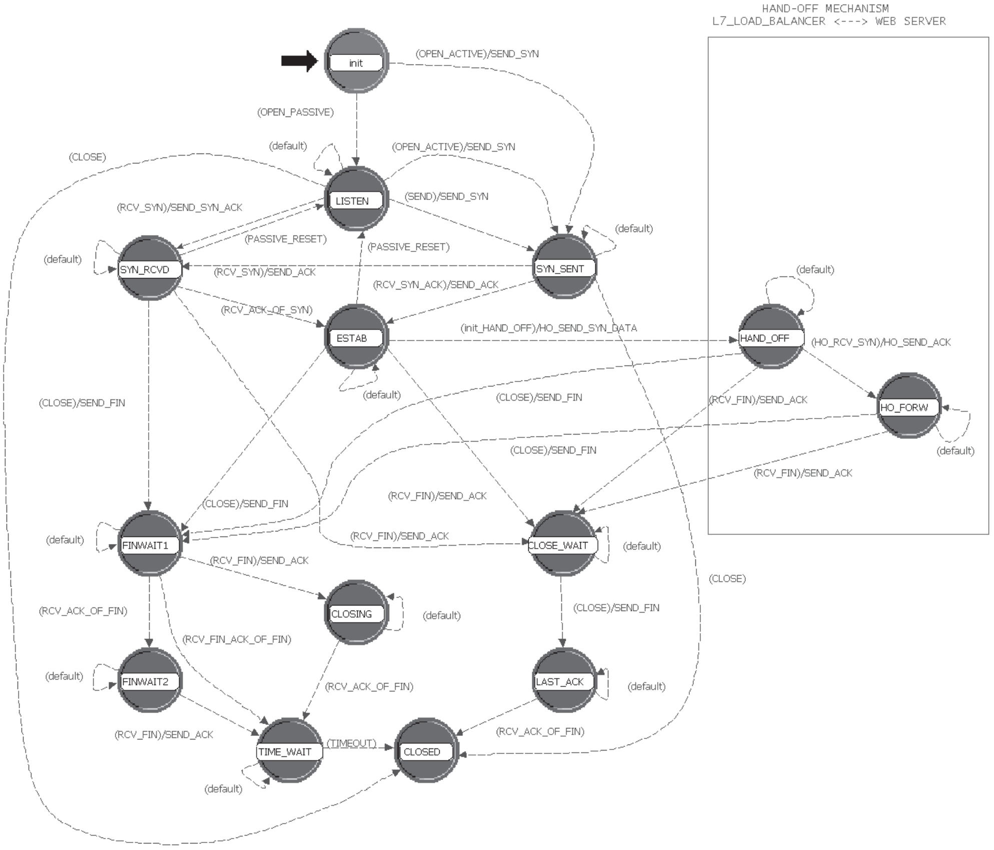


Figure A.4: Architecture of the simulation model

Acronyms

ALBM	Adaptive Load Balancing Mechanism
AON	Application Oriented Networking
App/DB	Application/Database
ARP	Address Resolution Protocol
AS	Autonomous System
BCB	Balanced Content-Based
CAD	Content-Aware Dispatching
CAHRD	Content-Aware Hybrid Request Distribution
CAP	Client-Aware Policy
CAWLL	Content-Aware Weighted Least Load
CPU	Central Processing Unit
CWARD/CR	Content-based Workload-Aware Request Distribution with Core Replication
CWARD/FR	Content-based Workload-Aware Request Distribution with Frequency-based Replication
CDNs	Content Distribution Networks
DNS	Domain Name System
DoS	Denial-of-Service

DR	Direct Routing
DSR	Direct Server Return
E-FSPF	Extended Fewest Server Processes First
FARD	Fuzzy Adaptive Request Distribution
HACC	Harvard Array of Clustered Computers
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
IPTun	IP Tunneling
ISS	Initial Send Sequence
IRS	Initial Receive Sequence
IQRD	Intelligent Queue-based Request Dispatcher
JSQ	Join Shortest Queue
KNITS	Knowledgeable Node Initiated TCP Splicing
LAN	Local Area Network
LARD	Locality-Aware Request Distribution
LC	Least Connection
LL	Least Loaded
LMS	Least Mean Square
LVS	Linux Virtual Server
L7SW	Linux Layer7 switching
MAA	Message-Aware Adaptive
MAC	Media Access Control

MSS	Maximum Segment Size
NAT	Network Address Translation
NLB	Network Load Balancing
NLMS	Normalised Least Mean Square
NPSSM	Non Probabilistic Server Selection Method
OS	Operative System
OSI	Open Systems Interconnection
TAP^2	Time and Access Probability-based Prefetch
TCP	Transmission Control Protocol
RAM	Random Access Memory
RR	Round Robin
RTT	Round-Trip Time
SAA	Session Affinity-Aware
SLA	Service Level Agreement
SLB	Server Load Balancing
SHLPN	Stochastic High-Level Petri Net
RXP	Resonate Exchange Protocol
QoS	Quality of Service
UDP	User Datagram Protocol
VIP	Virtual IP
VLAN	Virtual LAN
VOIP	Voice Over Internet Protocol

WAN	Wide Area Network
WARD	Workload-Aware Request Distribution
WLC	Weighted Least-Connection
WRR	Weighted Round Robin
xLARD/R	Extended Locality-Aware Request Distribution with Replication Policy

Bibliography

[1] Cisco systems, inc. http://www.cisco.com/.

[2] The internet traffic archive. http://ita.ee.lbl.gov/.

[3] OPNET technologies, inc. http://www.opnet.com/.

[4] Resonate, inc. http://www.resonate.com/.

[5] TCPHA project. http://dragon.linux-vs.org/ dragonfly/htm/tcpha.htm.

[6] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: a control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13:80–96, 2002.

[7] Jussara Almeida, Mihaela Dabu, and Pei Cao. Providing differentiated levels of service in web content hosting. In *proc. of the First Workshop on Internet Server Performance*, 1998.

[8] Mikael Andersson, Jianhua Cao, Maria Kihl, and Christian Nyberg. Admission control with service level agreements for a web server. In *proc. of European Internet and Multimedia Systems and Applications*, 2005.

[9] Mauro Andreolini, Michele Colajanni, and Marcello Nuccio. Kernel-based web switches providing content-aware routing. In *proc. of the 2nd IEEE International Symposium on Network Computing and Applications (NCA'03)*, 2003.

[10] George Apostolopoulos, David Aubespin, Vinod G. J. Peris, Prashant Pradhan, and Debanjan Saha. Design, implementation and performance of a content-based switch. In *proc. of INFOCOM*, 2000.

[11] Martin Arlitt and Tai Jin. A workload characterization of the 1998 World Cup Web site. Technical report hpl-1999-35r1, HP Labs, October 1999.

[12] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Efficient support for P-HTTP in cluster-based web servers. In *proc. of the annual conference on USENIX Annual Technical Conference*, 1999.

[13] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *proc. of ACM SIGMETRICS*, 2000.

[14] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *proc. of the USENIX 2000 Annual Technical Conference*, 2000.

[15] James Aweya, Michel Ouellette, Delfin Y. Montuno, Bernard Doray, and Kent Felske. An adaptive load balancing scheme for web servers. *International Journal of Network Management*, 12:3 – 39, 2002.

[16] Gaurav Banga and Peter Druschel. Measuring the capacity of a web server under realistic loads. In *proc. of World Wide Web*, 1999.

[17] Paul Barford and Mark Crovella. A performance evaluation of Hyper Text Transfer Protocols. In *proc. of ACM SIGMETRICS*, 1999.

[18] Luiz Barroso, Jeffrey Dean, and Urs Hoelzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23:22–28, 2003.

[19] Novella Bartolini, Giancarlo Bongiovanni, and Simone Silvestri. Self-* through self-learning: Overload control for distributed web systems. *Computer Networks*, 53:727–743, 2009.

[20] Yuliy Baryshnikov, Ed Coffman, Dan Rubenstein, and Teddy Yimwadsana. Traffic prediction on the internet. Technical report ee200514-1, Computer Networking Research Center Columbia Univeristy, May 2002.

[21] Leeann Bent, Michael Rabinovich, Geoffrey M. Voelker, and Zhen Xiao. Characterization of a large web site population with implications for content delivery. In *proc. of the 13th international conference on World Wide Web*, 2004.

[22] Nina Bhatti and Rich Friedrich. Web server support for tiered services. *IEEE Network*, September/October:64–71, 1999.

[23] Thomas Bonald and James W. Roberts. Congestion at flow level and the impact of user behaviour. *Computer Networks*, 42:521–536, 2003.

[24] Leszek Borzemski and Krzysztof Zatwarnicki. A fuzzy adaptive request distribution algorithm for cluster-based web systems. In *proc. of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro PDP)*, 2003.

[25] Juergen Brendel. Client-side resource-based load-balancing with delayed-resource-binding using TCP state migration to WWW server farm. United States Patent 6,182,139, January 2001. Resonate Inc.

[26] Thomas P. Brisco. DNS support for load balancing. RFC 1794, April 1995.

[27] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. *ACM Comput. Surv.*, 34:263–311, 2002.

[28] Valeria Cardellini, Michele Colajanni, and Phillip S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, May / June:28–39, 1999.

[29] Enrique Carrera and Ricardo Bianchini. Efficiency vs. portability in cluster-based network servers, 2001.

[30] Emiliano Casalicchio, Valeria Cardellini, and Michele Colajanni. Content-aware dispatching algorithms for cluster-basedweb servers. *Cluster Computing*, 5:65–74, 2002.

[31] Emiliano Casalicchio and Michele Colajanni. A client-aware dispatching algorithm for web clusters providing multiple services. In *proc. of the 10th international conference on World Wide Web*, 2001.

[32] Mohan Rao Cavale. *Introducing Microsoft Cluster Service (MSCS) in the Windows Server 2003 Family.* Microsoft Corporation, November 2002.

[33] Yeim-Kuan Chang, Wen-Hsin Cheng, and Chung-Ping Young. Fully pre-splicing TCP for web switches. In *proc. of the 1st International Conference on Innovative Computing, Information and Control (ICICIC)*, 2006.

[34] Huamin Chen and Prasant Mohapatra. Overload control in qos-aware web servers. *Computer Networks*, 42:119 – 133, 2003.

[35] Xiangping Chen, Huamin Chen, and Prasant Mohapatra. Aces: An efficient admission control scheme for qos-aware web servers. *Computer Communications*, 26:1581–1593, 2003.

[36] Sheng-Tzong Cheng, Chi-Ming Chen, and Ing-Ray Chen. Performance evaluation of an admission control algorithm: dynamic threshold with negotiation. *Performance Evaluation*, 52:1–13, 2003.

[37] Ludmila Cherkasova, Mohan DeSouza, and Shankar Ponnekanti. Performance analysis of "content-aware" load balancing strategy FLEX: Two case studies. In *proc. of the 34th Hawaii International Conference on System Sciences*, 2001.

[38] Ludmila Cherkasova and Magnus Karlsson. Scalable web server cluster design with workload-aware request distribution strategy WARD. In *proc. of the Third International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, 2001.

[39] Ludmila Cherkasova and Peter Phaal. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Trans. Computers*, 51:669–685, 2002.

[40] Mei-Ling Chiang, Yu-Chen Lin, Lian-Feng GuoThe Journal of Systems, and Software. Design and implementation of an efficient web cluster with content-based request distribution and file caching. *The Journal of Systems and Software*, 81:2044–2058, 2008.

[41] Eunmi Choi. Performance test and analysis for an adaptive load balancing mechanism on distributed server cluster systems. *Future Generation Computer Systems*, 20:237–247, 2004.

[42] Gianfranco Ciardo, Alma Riska, and Evgenia Smirni. EQUILOAD: a load balancing policy for clustered web servers. *Performance Evaluation*, 46(2-3):101–124, 2001.

[43] Inc. Cisco Systems. Scalable content switching. a discussion of the cisco css 11500 series content services switch architecture. White Paper, 2002.

[44] William S. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74:829–836, 1979.

[45] Ariel Cohen, Sampath Rangarajan, and Hamilton Slye. On the performance of TCP splicing for URL-aware redirection. In *proc. of the 2nd conference on USENIX Symposium on Internet Technologies and Systems*, 1999.

[46] Michele Colajanni and Philip S. Yu. A performance study of robust load sharing strategies for distributed heterogeneous web server systems. *Knowledge and Data Engineering*, 14(2):398–414, 2002.

[47] Steven Colby, John J. Krawezyk, Raj Krishnan Nair, Katherine Royee, Kenneth P. Siegel, Richard C. Stevens, and Seott Wasson. Method and system for directing a flow between a client and a server. United States Patent 6,006,264, December 2001. Arrowpoint Communications, Inc.

[48] Michael Dahlin. Interpreting stale load information. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1033–1047, 2000.

[49] Om P. Damani, Emerald Chung, Yennun Huang, Chandra Kintala, and Yi-Min Wang. ONE-IP: techniques for hosting a service on a cluster of machines. *Computer Networks and ISDN Systems*, 29:1019–1027, 1997.

[50] Defense Advanced Research Projects Agency (DARPA). Transmission control protocol. RFC 793, September 1981.

[51] Lars Eggert and John Heidemann. Application-level differentiated services for web servers. *World Wide Web*, 2:133 – 142, 1999.

[52] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *proc. of the 13th international conference on World Wide Web*, 2004.

[53] F5 Networks, Inc. http://www.f5.com/.

[54] Ahmad Faour and Nashat Mansour. Weblins: A scalable www cluster-based server. *Advances in Engineering Software*, 37:11–19, 2006.

[55] Roy T. Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, L. Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol - HTTP/1.1. *RFC*, 2616, June 1999.

[56] Xuehong Gan, Trevor Schroeder, Steve Goddard, and Byrav Ramamurthy. Highly available and scalable cluster-based web servers. In *proc. of the 8th IEEE International Conference on Computer Communications and Networks*, 1999.

[57] Rosario G. Garroppo, Stefano Giordano, Michele Pagano, and Gregorio Procissi. On traffic prediction for resource allocation: A chebyshev bound based allocation scheme. *Computer Communications*, 31:3741–3751, 2008.

[58] Mario Gerla, M. Y. Sanadidi, Ren Wang, and Andrea Zanella. TCP westwood: Congestion window control using bandwidth estimation. In *proc. of IEEE GLOBECOM*, 2001.

[59] Katja Gilly, Salvador Alcaraz, Carlos Juiz, and Ramon Puigjaner. Comparison of predictive techniques in cluster-based network servers with resource allocation. In *proc. of the 12th Annual Meeting of the IEEE / ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2004.

[60] Katja Gilly, Salvador Alcaraz, Carlos Juiz, and Ramon Puigjaner. A predictive-adaptive algorithm for a web switch. In *proc. of the 8th World Scientific and Engineering Academy and Society International Multiconference. Circuits, Systems, Communications & Computers (CSCC)*, 2004.

[61] Katja Gilly, Salvador Alcaraz, Carlos Juiz, and Ramon Puigjaner. Prediction accuracy study of an adaptive algorithm in cluster based network servers with resource allocation. In *proc. of ISCIS. Advances in Computer Science and Engineering: Reports - Vol. 1. New Trends in Computer Networks*, 2005.

[62] Katja Gilly, Salvador Alcaraz, Carlos Juiz, and Ramon Puigjaner. Resource allocation study based on burstiness for a web switch. In *proc. of the 4th International Information and Telecommunication Technologies Symposium (I2TS)*, 2005.

[63] Katja Gilly, Salvador Alcaraz, Carlos Juiz, and Ramon Puigjaner. Algoritmo predictivo-adaptatitvo para conmutadores en agrupaciones de servidores web. *IEEE Latin American Transactions*, 4:62–68, 2006.

[64] Katja Gilly, Salvador Alcaraz, Carlos Juiz, and Ramon Puigjaner. An implementation of a layer-7 load balancer in OPNET Modeler. In *OPNETWORK*, 2006.

[65] Katja Gilly, Salvador Alcaraz, Carlos Juiz, and Ramon Puigjaner. Service differentiation and QoS in a scalable content-aware load balancing algorithm. In *proc. of the 40th Annual Simulation Symposium*, 2007.

[66] Katja Gilly, Salvador Alcaraz, Carlos Juiz, and Ramon Puigjaner. Burstiness detection in a web adaptive cluster system. In *proc. of the Industry Track to First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTOOLS)*, 2008.

[67] Katja Gilly, Salvador Alcaraz, Carlos Juiz, and Ramon Puigjaner. Analysis of burstiness monitoring and detection in an adaptive web system. *Computer Networks*, 53:668–679, 2009.

[68] Katja Gilly, Javier Huertas, Carlos Juiz, and Ramon Puigjaner. Burstiness detection location in cluster-based network servers based on the APRA algorithm: a case study. In *proc. of IADIS International Conference WWW/Internet*, 2006.

[69] Katja Gilly, Carlos Juiz, Salvador Alcaraz, and Ramon Puigjaner. Adaptive admission control algorithm in a QoS-aware web system. In *proc. of IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS) - to appear*, 2009.

[70] Katja Gilly, Carlos Juiz, Ramon Puigjaner, and Salvador Alcaraz. Performance analysis of a predictive - adaptive algorithm in cluster-based network servers with resource allocation. In *proc. of the 19th International Symposium of Computer and Information Sciences (ISCIS)*, 2004.

[71] Katja Gilly, Carlos Juiz, and Nigel Thomas. Scalable QoS content-aware load balancing algorithm for a cluster based network web servers. In *UK Performance Engineering Workshop*, 2007.

[72] Katja Gilly, Nigel Thomas, Carlos Juiz, and Ramon Puigjaner. Scalable QoS content-aware load balancing algorithm for a web switch based on classical policies. In *proc. of 22nd International Conference on Advanced Information Networking and Applications (AINA)*, 2008.

[73] Steve Goddard and Trevor Schroeder. The SASHA architecture for network-clustered web servers. In *proc. of the 6th IEEE International Symposium on High Assurance Systems Engineering*, 2001.

[74] Graham C. Goodwin and Kwai Sang Sin. *Adaptive Filtering: Prediction and Control.* Prentice-Hall, 1984.

[75] Riccardo Gusella. Characterizing the variability of arrival processes with indexes of dispersion. *IEEE Journal on Selected Areas in Communications*, 9:203–211, 1991.

[76] Simon Haykin. *Adaptive Filter Theory.* Prentice-Hall, 1991.

[77] Guerney Hunt, Erich Nahum, and John Tracey. Enabling content-based load distribution for scalable services. Technical report, IBM T.J. Watson Research Center, May 1997.

[78] IBM. http://www.ibm.com/.

[79] IBM. Application switching with nortel networks layer 2-7 gigabit ethernet switch module for ibm bladecenter. IBM Redbook, March 2006.

[80] Arun Iyengar, Jim Challenger, Daniel Dias, and Paul Dantzig. High-performance web site design techniques. *IEEE Internet Computing*, Volume 4:17 – 26, 2000.

[81] Christoforos Kachris and Stamatis Vassiliadis. Design of a web switch in a reconfigurable platform. In *proc. of the 2006 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2006.

[82] Vikram. Kanodia and Edward W. Knightly. Multi-class latency-bounded web services. In *proc. of te 8th International Workshop onQuality of Service (IWQOS)*, 2000.

[83] Daorat Kerdlapanan and Akharin Khunkitti. Content-based load balancing with multicast and tcp-handoff. In *proc. of International Symposium on Circuits and Systems*, 2003.

[84] Maria Kihl, Anders Robertsson, Mikael Andersson, and Björn Wittenmark. Control-theoretic analysis of admission control mechanisms for web server systems. *World Wide Web*, 11:93–116, 2008.

[85] Maria Kihl and Niklas Widell. Admission control schemes guaranteeing customer QOS in commercial web sites. In *proc. of the IFIP TC6 / WG6.2 & WG6.7 Conference on Network Control and Engineering for QoS, Security and Mobility*, 2002.

[86] Masayoshi Kobayashi and Tutomu Murase. Asymmetric tcp splicing for content-based switches. In *proc. of IEEE International Conference on Communications(ICC)*, 2002.

[87] Ravi Kokku, Ramakrishnan Rajamony, and Lorenzo Alvisi andHarrick Vin. Half-pipe anchoring: an efficient technique for multiple connection handoff. In *proc. of the 10th IEEE International Conference on Network Protocols*, 2002.

[88] Chandra Kopparapu. *Load balancing Servers, Firewalls and Caches.* Wiley, 2001.

[89] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. On the use and performance of content distribution networks. In *proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 2001.

[90] Kun-Chan Lan and John Heidemann. A measurement study of correlations of internet flow characteristics. *Computer Networks*, 50(1):46–62, January 2006.

[91] Chang Li, Gang Peng, Kartik Gopalan, and Tzi cker Chiueh. Performance guarantee for cluster-based internet services. In *proc. of the 23rd International Conference on Distributed Computing Systems*, 2003.

[92] Houjin Li, Changcheng Huang, Michael Devetsikiotis, and Gerard Damm. Effective bandwidths under dynamic weighted round robin scheduling. In *proc. of GLOBE-COM*, 2004.

[93] Ying-Dar Lin, Ping-Tsai Tsai, Po-Ching Lin, and Ching-Ming Tien. Direct web switch routing with state migration, TCP masquerade, and cookie name rewriting. In *proc. of Global Telecommunications Conference*, 2003.

[94] Ho-Han Liu and Mei-Ling Chiang. Tcp rebuilding for content-aware request dispatching in web clusters. *Journal of Internet Technology*, 6:231–240, 2005.

[95] Ho-Han Liu, Mei-Ling Chiang, and Men-Chao Wu. Efficient support for content-aware request distribution and persistent connection in Web clusters. *Software: Practise and Experience*, 37:1215–1241, 2007.

[96] Mon-Yen Luo and Chu-Sing Yang. System support for scalable, reliable and highly manageable web hosting service. In *proc. of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, 2001.

[97] Mon-Yen Luo, Chu-Sing Yang, and Chun-Wei Tseng. Analysis and improvement of content-aware routing mechanisms. *IEICE Transactions on Communications*, E88:227–238, 2005.

[98] David A. Maltz and Pravin Bhagwat. TCP splicing for application layer proxy performance. Technical report, IBM, 1998.

[99] Manish Marwah, Shivakant Mishra, and Christof Fetzer. Fault-tolerant and scalable TCP splice and web server architecture. In *proc. of the 25th IEEE Symposium on Reliable Distributed Systems*, 2006.

[100] Mark Meiss, Filippo Menczer, and Alessandro Vespignani. On the lack of typical behavior in the global web traffic network. In *proc. of World Wide Web*, 2005.

[101] Daniel A. Menascé and Virgilio A.F. Almeida. *Capacity Planning for Web Performance. Metrics, Models and Methods.* Prentice Hall, 1998.

[102] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Burstiness in multi-tier applications: symptoms, causes, and new models. In *proc. of the 9th ACM/IFIP/USENIX International Conference on Middleware*, 2008.

[103] Jeffrey C. Mogul. The case for persistent-connection HTTP. In *proc. of SIGCOMM*, 1995.

[104] Foundry Networks. http://www.foundrynet.com.

[105] Nortel Networks. http://www.nortel.com/.

[106] Minhwan Ok and Myong-Soon Park. Distributing requests by (around k)-bounded load-balancing in web server cluster with high scalability. *IEICE-Transactions on Informations and Systems*, E89-D:663–672, 2006.

[107] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. Locality-aware request distribution in cluster-based network servers. In *proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[108] Raju Pandey, J. Fritz, and Barnes Ronald Olsson. Supporting quality of service in http servers. In *proc. of the 7th Annual SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1998.

[109] Athanasios E. Papathanasiou and Eric Van Hensbergen. KNITS: switch-based connection hand-off. In *proc. of IEEE INFOCOM*, 2002.

[110] Seon-Yeong Park, Dohyun Park, Joonwon Lee, and Jung Wan Cho. Efficient inter-backend prefetch algorithms in cluster-based web servers. In *proc. of International Conference/Exhibition on High Performance Computing*, 2001.

[111] Nicolas Poggi, Toni Moreno, Josep Lluis Berral, Ricard Gavaldà, and Jordi Torres. Self-adaptive utility-based web session management. *Computer Networks*, 53:1712–1721, 2009.

[112] Radware. http://www.radware.com.

[113] Resonate. Resonate central dispatch technology advantage: TCP connection HOP. White Paper, April 2001.

[114] Alma Riska, Wei Sun, Evgenia Smirni, and Gianfranco Ciardo. ADAPTLOAD: effective balancing in clustered web servers under transient load conditions. In *proc. of the 22nd International Conference on Distributed Computing Systems*, 2002.

[115] Joseph Lee Rodgers and W. Alan Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42:59–66, 1988.

[116] Marcel-Catalin Rosu and Daniela Rosu. An evaluation of TCP splice benefits in web proxy servers. In *proc. of WWW*, 2002.

[117] Shriram Sarvotham, Rudolf Riedi, and Richard Baraniuk. Connection-level analysis and modeling of network traffic. In *proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 2001.

[118] Shinsuke Satake and Hiroshi Inai. Special issue on internet architecture technology papers: A nonprobabilistic server selection method based on periodically obtained load information for web server clusters. *Electronics and Communications in Japan*, 89:1–12, 2006.

[119] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Transactions on Internet Technology*, 6:20–52, 2006.

[120] Trevor Schroeder, Steve Goddard, and Byrav Ramamurthy. Scalable web server ciustering technologies. *IEEE Network*, May:38–46, 2000.

[121] Linux Virtual Server. http://www.linuxvirtualserver.org/, 2006.

[122] Zhiguang Shan, C. Lin, D. C. Marinescu, and Y. Yang. Modeling and performance analysis of QoS-aware load balancing of web-server clusters. *Computer Networks*, 40:235–256, 2002.

[123] Saeed Sharifian, Seyed A. Motamedi, and Mohammad K. Akbarib. A content-based load balancing algorithm with admission control for cluster web servers. *Future Generation Computer Systems*, 24:775–787, 2008.

[124] Yiu-Fai Sit, Cho-Li Wang, and Francis Lau. Socket cloning for cluster-based web servers. In *proc. of IEEE International Conference on Cluster Computing*, 2002.

[125] Yiu-Fai Sit, Cho-Li Wang, and Francis Lau. Cyclone: A high-performance cluster-based web server with socket cloning. *Cluster Computing*, 7:21–37, 2004.

[126] Alex C. Snoeren, David G. Andersen, and Hari Balakrishnan. Fine-grained failover using connection migration. In *proc. of 3rd USENIX Symp. on Internet Technologies and Systems*, 2001.

[127] Linux Layer7 Switching. http:/www.linux-l7sw.org/.

[128] Matthew Syme and Pilip Goldie. *Optimizing Network Performance with content switching. Server, Firewall and Cache Load Balancing.* Prentice Hall, 2004.

[129] Masahiko Takahashi, Akihito Kohiga, Tomoyoshi Sugawara, and Atsuhiro Tanaka. Tcp-migration with application-layer dispatching: A new http request distribution

architecture in locally distributed web server systems. In *proc. of the 1st International Conference on Communication System Software and Middleware*, 2006.

[130] Wenting Tang, Ludmila Cherkasova, Lance Russell, and Matt W. Mutka. Modular tcp handoff design in streams-based tcp/ip implementation. In *proc. of the 1st International Conference on Networking-Part 2*, 2001.

[131] Xueyan Tang and Samuel T. Chanson. On caching effectiveness of web clusters under persistent connections. *Journal of Parallel and Distributed Computing*, 63:981 – 995, 2003.

[132] Yong Meng Teo and Rassul Ayani. Comparison of load balancing strategies on cluster-based web servers. *Transactions of the Society for Modeling and Simulation*, 77:185–195, 2001.

[133] Bhuvan Urgaonkar and Prashant Shenoy. Cataclysm: Scalable overload policing for internet applications. *Journal of Network and Computer Applications*, 31:891– 920, 2008.

[134] Remco van de Meent, Aiko Pras, Michel Mandjes, J.L. van den Berg, F. Roijers, Lambert J. M. Nieuwenhuis, and Pieter H.A. Venemans. Burstiness predictions based on rough network traffic measurements. In *proc. of 19th World Telecommunications Congress (WTC/ISS)*, 2004.

[135] Michail Vlachos, Chris Meek, Zografoula Vagena, and Dimitrios Gunopulos. Identifying similarities, periodicities and bursts for online search queries. In *proc. of SIGMOD*, 2004.

[136] Thiemo Voigt and Per Gunningberg. Adaptive resource-based web server admission control. In *proc. of the 7th International Symposium on Computers and Communications*, 2002.

[137] Zheng Wang and Jon Crowcroft. Analysis of burstiness and jitter in real-time communications. In *proc. of SIGCOMM*, 1993.

[138] David X. Wei and Steven H. Low. A burstiness control for TCP. In *proc. of the Workshop on Protocols for Fast Long-Distance Networks*, 2005.

[139] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *proc. of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, 2003.

[140] Bernard Widrow, Jr. John .R. Glover, John M. McCool, John Kaunitz, Charles S. Williams, Robert H. Hearn, James R. Zeidler, Jr. Eugene Dong, and Robert C. Goodlin. Adaptive noise cancelling: Principles and applications. In *proc. of IEEE*, 1975.

[141] Chu-Sing Yang and Mon-Yen Luo. Efficient support for content-based routing in web server clusters. In *proc. of the 2nd conference on USENIX Symposium on Internet Technologies and Systems - Volume 2*, 1999.

[142] Jianhua Yang, Di Jin, Ye Li, Kai-Steffen Hielscher, and Reinhard German. Modeling and simulation of performance analysis for a cluster-based web server. *Simulation Modelling Practice and Theory*, 14:188–200, 2006.

[143] Jingnan Yao, Jianxun Jason Ding, and Laxmi N. Bhuyan. Intelligent message scheduling in application oriented networking systems. In *proc. of IEEE International Conference on Communications (ICC)*, 2008.

[144] Du Zeng-Kai and Ju Jiu-Bin. A completely distributed architecture for cluster-based web servers. In *proc. of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2003.

[145] Qi Zhang, Ningfang Mi, Alma Riska, and Evgenia Smirni. Load unbalancing to improve performance under autocorrelated traffic. In *proc. of the 26th IEEE International Conference on Distributed Computing Systems*, 2006.

[146] Qi Zhang, Alma Riska, and Erik Riedel. Workload propagation overload in bursty servers. In *proc. of the 2nd International Conference on the Quantitative Evaluation of Systems (QEST)*, 2005.

[147] Qi Zhang, Alma Riska, Wei Sun, Evgenia Smirni, and Gianfranco Ciardo. Workload-aware load balancing for clustered web servers. *IEEE Transactions on Parallel and Distributed Systems*, 3:219–233, 2005.

[148] Ronghua Zhang, Tarek F. Abdelzaher, and John A. Stankovic. Efficient TCP connection failover in web server clusters. In *proc. of IEEE INFOCOM*, 2004.

[149] Wensong Zhang. Linux virtual server for scalable network services. In *proc. of OTTAWA Linux Symposium*, 2000.

[150] Xiaolan Zhang, Michael Barrientos, J. Bradley Chen, and Margo Seltzer. HACC: An architecture for cluster-based web servers. In *proc. of the 3rd USENIX Windows NT Symposium*, 1999.

[151] Li Zhao, Yan Luo, Laxmi Bhuyan, and Ravi Iyer. Design and implementation of a content-aware switch using a network processor. In *proc. of the 13th Symposium on High Performance Interconnects*, 2005.

[152] Jingyu Zhou and Tao Yang. Selective early request termination for busy internet services. In *proc. of the 15th International Conference on World Wide Web*, 2006.